UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELETRICA

DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇAO E AUTOMAÇAO INDUSTRIAL

ESTRUTURAÇAO DE PROGRAMAS PARA SUPORTE A REENGENHARIA

Por : Maria Fernanda Moura

Orientador : Prof. Dr. Mário Jino

Co-Orientador : Prof. Dr. Fuad Gattaz Sobrinho

DISSERTAÇAO APRESENTADA A FACULDADE DE ENGENHARIA ELETRICA, DA UNIVERSIDADE ESTADUAL DE CAMPINAS, COMO REQUISITO PARCIAL PARA OBTENÇAO DO TITULO DE MESTRE EM ENGENHARIA ELETRICA.

DEZEMBRO 1992

A Janda e ao Beto

AGRADECIMENTOS

Ao Prof. Dr. Mário Jino pela orientação, compreensão e apoio recebidos;

Ao Prof. Dr. Fuad Gattaz Sobrinho cujo trabalho inicial originou o aqui presente, e, pelas discussões e apoio recebidos;

Ao Sr. Alvaro Seixas Neto, chefe do NTIA/EMBRAPA, pelo empenho em manter o "programa informal de pós-graduação", oportunidade sem a qual este trabalho não existiria;

Ao meu companheiro de "luta", Marcos Lordello Chaim, pelo desenvolvimento e implementação do protótipo COB_FNC, pelas sugestões e apoio constantes;

Ao meu amigo Mauro Carnassale pelas discussões e sugestões iniciais;

A Plínio de Sá Leitão Jr. pelas discussões e pelos grafos gerados com o módulo "chanomat" da ferramenta POKE-TOOL, versão COBOL;

A Marcélia Rezende, Carla Geovana do Nascimento Macário e Mário Antonio do Nascimento pela amizade, colaboração e apoio sempre presentes;

E, ao meu "amigo", Roberto Hiroshi Higa, pelas "bolinhas", apoio, carinho e compreensão sempre presentes.

RESUMO

Este trabalho propõe um Modelo de Estruturação baseado na decomposição de programas em programas primos, unidades mínimas e básicas da programação estruturada.

O modelo é totalmente definido por algoritmos e pode ser completamente automatizado, tratando códigos fonte de produtos de software em qualquer linguagem procedimental e não paralela. Particularmente, são dados os algoritmos e transformações necessárias para instanciar o modelo para as linguagens de programação COBOL, FORTRAN e C.

O modelo obtém uma representação padronizada do código fonte original, composta por unidades de programa (subprogramas), onde existem apenas um único ponto de entrada e um único ponto de saída, e são utilizados apenas três tipos de estruturas de controle : comandos seqüenciais, um único tipo de iteração e seleções simples (de duas saídas).

A padronização do código fonte é obtida por uma linguagem intermediária, executável e facilmente portável - um subconjunto da linguagem de programação C.

O padrão de representação do código estruturado pelo modelo fornece uma base adequada, mínima, necessária e suficiente para servir de suporte a um ambiente de ferramentas de reengenharia e/ou engenharia reversa. A própria implementação do modelo é classificada como uma ferramenta de reengenharia.

Os protótipos implementados, assim como o modelo de implementação, foram desenvolvidos em um ambiente UNIX.

ABSTRACT

A Structuring Model, based on the decomposition of programs into prime programs, is proposed . Prime programs are minimal and basic units of structured programming.

The model is completely defined by algorithms and can be automatized to deal with source code of software products implemented in any procedural non-parallel programming language. Algorithms and necessary transformations to instantiate the model for COBOL, FORTRAN and C programming languages are given.

The model produces a standardized representation from the source code, compounded by program units, with a single entry point and a single exit point, composed by three types of control structures only : sequencial statements, an unique type of iteractive structure and single selections.

The standardized representation of the source code is provided by an executable portable intermediate language - a sub set of the C programming language.

The standardized representation of the structured source code provides a minimal, necessary and sufficient base to support re-engineering and / or reverse engineering environments. The model implementation by itself is classified as a re-engineering tool.

The implemented prototypes and the model of implementation were developed in a UNIX environment.

INDICE

Capítulo 1 : Introdução 001

1.1. Contexto do Trabalho 002

1.2. Revisão Bibliográfica 005

1.3. Objetivo e Organização do Trabalho 008

Capítulo 2 : Definições Básicas e O Modelo de Estruturação 010

2.1. Definições Básicas 010

2.1.1. Programas Próprios e Programas Primos 010

2.1.2. Teorema da Estrutura e Primeira Forma Normal 015

2.1.3. Simplificação da Primeira Forma Normal 021

2.1.4. Estruturas de Controle e Seqüenciamento ... 023

2.1.5. Estrutura de Representação - PrimeTree 027

2.2. O Modelo de Estruturação 031

Capítulo 3 : Esquemas de Decomposição - Linguagens Procedimentais 035

3.1. Esquema de Decomposição para a Linguagem COBOL ... 039

3.2. Esquema de Decomposição para a Linguagem FORTRAN .. 053

3.3. Esquema de Decomposição para a Linguagem C 062

Capítulo 4 : Redução da Estrutura do Programa 066

4.1. O Modelo de Redução 068

4.2. Exemplo de Aplicação 075

Capítulo 5 : Implementação do Modelo de Estruturação e Exemplo de Aplicação 084

5.1. Implementação do Modelo de Estruturação 084

5.1.1. Protótipo Implementado 086

5.2. Exemplo de Aplicação dos Protótipos 095

5.2.1. Estruturação do Módulo SPLINE/NTIA 096

5.2.2. Número de Linhas de Código
do Módulo SPLINE/NTIA 097

5.2.3. Equivalência Funcional 100

5.2.4. Análise da Estrutura Gerada
versus Original 102

5.2.5. Tratamento de Código Morto 107

Capítulo 6 : Conclusões 110

6.1. Contribuições do Modelo 110

6.2. Trabalhos Futuros 111

6.3. Considerações Finais 115

Bibliografia 116

Apêndices

A. Seqüenciamentos para as Estruturas de Controle
da Linguagem COBOL 121

B. Seqüenciamentos para as Estruturas de Controle
da Linguagem FORTRAN 127

C. Seqüenciamentos para as Estruturas de Controle
da Linguagem C 134

D. Referências ao Módulo SPLINE/NTIA
- Original e Estruturado 143

Lista de Figuras

Figura 1.1 : Modelo do Sistema de Reengenharia 004
Figura 2.1 : Nó Funcional, Nó Predicativo e Nó Coletor 011
Figura 2.2 : Programas Próprios e Não Próprios 013
Figura 2.3 : Programas Primos 014
Figura 2.4 : Fluxograma Exemplo 016
Figura 2.5 : Seqüências para Fluxograma Exemplo 019
Figura 2.6 : Fluxograma Exemplo Seqüenciado
- Teorema da Estrutura 020

Figura 2.7 : Resultado da Eliminação de Testes e Atribuições
Desnecessários 022
Figura 2.8 : Seqüências na PrimeTree 028
Figura 2.9 : PrimeTree
- Primeira Forma Normal do Exemplo 029
Figura 2.10 : Modelo de Estruturação Proposto 032
Figura 3.1 : PrimeTree do Pseudo-Código
- Primeira Forma Normal 036
Figura 3.2 : PrimeTree -
Seqüenciamento do Programa C com 3 "returns" 038
Figura 3.3 : Grafo de Fluxo de Controle - Exemplo -
- Código COBOL 041
Figura 3.4 : Esquema de Decomposição Generalizado - COBOL 043
Figura 3.5 : Algoritmo para Construir P_1' e a Tabela D_1
- Fase 2 - COBOL 048
Figura 3.6 : Algoritmo para Tratamento de Desvios
- Fase 3 - COBOL 051
Figura 3.7 : Algoritmo para Obter P_{1k}' e a Tabela D_{11}
- Fase 2 - FORTRAN 058
Figura 3.8 : Algoritmo para Tratamento de Desvios
- Fase 3 - FORTRAN 061
Figura 4.1 : Modelo de Redução 067
Figura 4.2 : Composição de Duas Seqüências
- Corpo de Seqüência 068
Figura 4.3 : Fatoração de um Nó do Tipo OR
- com duas saídas iguais 069
Figura 4.4 : Algoritmo EliminaSaida 070
Figura 4.5 : Algoritmo CompoeElimina 071
Figura 4.6 : Algoritmo FatoraOrNode 071
Figura 4.7 : Seqüência Recursiva
Equivalente a um Nó do Tipo LOOP 073
Figura 4.8 : Seqüência Recursiva sem Predicativo
- "Loop Infinito" 073
Figura 4.9 : Algoritmos para Tratamento de "Loops" 074
Figura 4.10 : Primeira Forma Normal
da Unidade de Programa "d" 076
Figura 4.11 : PrimeTree "d" ao Final da Quinta Iteração
de "CompoeElimina" 077
Figura 4.12 : Fatoração da Seqüência s_{17} - PrimeTree "d" 077
Figura 4.13 : PrimeTree "d" ao Final da Décima Quarta Iteração
de "CompoeElimina" 078
Figura 4.14 : "IdentificaLoops" - PrimeTree "d" 079
Figura 4.15 : Composição de s_{12} com s_{22}
- PrimeTree "d" 080
Figura 4.16 : "IdentificaLoops" - Seqüência s_{12}
- PrimeTree "d" 082

Figura 4.17 : PrimeTree "d" Reduzida
- Forma Normal Reduzida 083
Figura 5.1 : Modelo de Implementação
de um Esquema de Decomposição 085
Figura 5.2 : Plano de Implementação do Modelo de Estruturação ... 087
Figura 5.3 : Plano de Implementação da Ferramenta COB_FNC 088
Figura 5.4 : Protótipos Implementados 088
Figura 5.5 : Arquitetura do Protótipo da Ferramenta C_FNC 090
Figura 5.6 : Arquitetura do Protótipo da Ferramenta REDUCAO 092
Figura 5.7 : Grafo de Fluxo de Controle - Função "token_var"
- Módulo SPLINE/NTIA 104
Figura 5.8 : Grafo de Fluxo de Controle - Função "semantica"
- Módulo SPLINE/NTIA 105

Lista de Tabelas

Tabela 2.1 : Teorema da Estrutura
Aplicado às Estruturas de Controle 024
Tabela 2.2 : Tabela de Transformações 026
Tabela 3.1 : Tabela de Rótulos R_0 046
Tabela 3.2 : Tabela de Desvios D_1 050
Tabela 3.3 : Tabela de "Entry-Points" para $P_2' - E_{20}$ 057
Tabela 3.4 : Tabela de Desvios para P_{2k}"- D_{21} 059
Tabela 5.1 : NLOC para as Funções do Arquivo "getpar.c" 098
Tabela 5.2 : NLOC para as Funções do Arquivo "lerobs.c" 099
Tabela 5.3 : NLOC para as Funções do Arquivo "outhdr.c" 099
Tabela 5.4 : NLOC para as Funções do Arquivo "semantic.c" 099
Tabela 5.5 : NLOC para as Funções do Arquivo "spline.c" 099
Tabela 5.6 : Valores Calculados para Interpolação
pelo Módulo Original e pelo Módulo Estruturado 101
Tabela 5.7 : Medida Ciclomática - Funções Estruturadas no Código
Fonte Original - Módulo SPLINE/NTIA 103
Tabela 5.8 : Medida Ciclomática - Funções não Estruturadas no
Código Fonte Original - Módulo SPLINE/NTIA 106

1 - INTRODUÇÃO

O problema atual de produção de software pode ser resumido ao termo "crise de software". Crise de software é a conseqüência da escassez de técnicas eficientes para a produção de software de qualidade, o que implica em baixa produtividade, em contraposição ao desenvolvimento e produção de hardware que caminham a passos rápidos e a custos baixos.

O custo da produção de software é elevado; 80 % de todo o orçamento é destinado à manutenção dos sistemas existentes, coexistente com grande demanda de novos produtos [YEH, 90-a].

A evolução (e/ou manutenção) do arsenal de software existente tende a levar a um aumento de complexidade, refletindo o deterioramento estrutural e, conseqüentemente, aumentando e dificultando o trabalho de manutenção [LEHMAN, 80]. A indústria norte-americana por si só possui US$ 300 bilhões em software mal-estruturado e de difícil manutenção [YEH, 90-a].

Propostas para reduzir o custo de produção de software, especialmente na fase de manutenção, levam a técnicas de reuso. Produzir software voltado para reuso é uma solução para baratear custos futuros, pois implica em desenvolver novos produtos ou redesenvolver produtos existentes [CALDIERA, 90].

Uma proposta para tornar reusável o arsenal de software existente é aplicar técnicas de engenharia reversa e/ou reengenharia, procurando recuperar informações sobre o produto e/ou reconstruí-lo.

Engenharia reversa é o processo de análise de um dado sistema de software, buscando identificar componentes e suas inter-relações e criando representações do sistema em outras formas ou em um nível de abstração mais alto [CHIKOFSKY, 90].

Reengenharia é o exame e alteração de um dado sistema de software para reconstruí-lo em uma nova forma e a sua subseqüente implementação sob a nova forma [CHIKOFSKY, 90].

A engenharia reversa permite recuperar informações sobre o projeto do produto, que podem ser usadas para construir um novo produto ou auxiliar processos de reengenharia automática, ou semi-automática, para reusar partes ou o todo do código [CALDIERA, 90].

A reengenharia voltada para reuso de um produto de software implica em mudar a forma do software existente, o que envolve técnicas de "cleanning-up", como estruturação do fluxo de controle, remoção de parâmetros supérfluos e de efeitos colaterais de variáveis. O objetivo é reduzir o custo de manutenção, aumentar o potencial de reuso, aumentar o tempo de vida e a qualidade do software [ARNOLD, 89].

Portanto, reengenharia e engenharia reversa aplicadas em conjunto sobre um produto de software, auxiliam a transformação do padrão de desenvolvimento e a compreensão do produto, diminuindo o custo de

manutenção e permitindo até reuso de projeto ou mesmo código [CALDIERA, 90].

Toda proposta de reengenharia e engenharia reversa passa por técnicas de "cleanning-up", sobre o produto de software original, cujo primeiro passo é a padronização das estruturas de controle presentes no código fonte do produto [ARNOLD, 89]. Ainda, dentro do escopo de engenharia reversa, um conjunto de informações importantes para a gerência de manutenção de sistemas de software são as métricas de complexidade de mudança [GATTAZ, 84].

O esforço de obtenção de métricas de complexidade estrutural e complexidade de esforço de mudança (medidas entre módulos do software) passa obrigatoriamente pela padronização das estruturas do fluxo de controle do sistema de software analisado ([GATTAZ, 84], [FENTON, 86] e [FENTON, 87]). Muitos métodos de estruturação de programas procuram padronizá-los para melhorar sua cognitividade ou representar os programas em uma linguagem estruturada ([BOHM, 66], [ASHCROFT, 71], [HOPKINS, 72], [WULF, 72], [WIRTH, 74], [URSCHLER, 75], [BAKER, 72], [KNUTH, 77], [DIJKSTRA, 79] e [CREAK, 87]).

Quando a padronização é o suporte para ferramentas de engenharia reversa e/ou reengenharia a preocupação é obter uma padronização matematicamente modulável [LINGER, 79], na qual possa ser baseado um processo de abstração para compreensão do comportamento do código ([BASILI, 82], [HAUSLER, 90], [CALDIERA, 90] e [EVANGELISTA, 92]).

O objetivo deste trabalho é obter um modelo sistemático para a estruturação de programas, que possa ser automatizado e cujo resultado forneça uma base adequada para ferramentas de reengenharia e/ou engenharia reversa.

Ao ser aplicado ao código fonte de um produto de software, implementado em uma linguagem procedimental, o modelo deve fornecer uma representação padronizada e funcionalmente equivalente ao código original.

Particularmente, o modelo deve satisfazer aos requisitos do **Projeto de Reengenharia** [MACARIO, 92], apresentado na próxima seção, onde a aplicação da ferramenta derivada do modelo é o passo inicial.

1.1. Contexto do Trabalho

O **Projeto de Reengenharia**[1] visa transformar o padrão de desenvolvimento do produto de software, que lhe é fornecido como entrada e cujo código fonte esteja em uma linguagem procedimental e não paralela, para um novo padrão, sob o paradigma de componentes ([MACARIO, 92]).

---

[1] *O Projeto de Reengenharia está sendo desenvolvido no NTIA/EMBRAPA [MACARIO, 92].*

O paradigma de componentes é a visão de que um software é uma composição de elementos abstratos e independentes, que podem ser reusados por vários produtos de software [MACARIO, 91].

O modelo do **Sistema de Reengenharia**, que pode ser observado na Figura 1.1, compõe-se das seguintes etapas :

a. Estruturação do Código Fonte :

Esta etapa recebe como entrada o código fonte de um produto de software e fornece uma representação padronizada do mesmo, representado através de um subconjunto da linguagem **C**.

A padronização implica em obter unidades de programa[2], a partir do código fonte original. As unidades são caracterizadas por um único ponto de entrada e um único ponto de saída, e compostas somente por comandos seqüenciais, iterações de um único tipo e seleções simples (de duas saídas).

b. Identificação de Partições :

Nesta etapa, a partir do código fonte padronizado e através de ferramentas de análise de interdependência de variáveis ([REZENDE, 92-a] e [REZENDE, 92-b]), são identificadas partições do código estruturado.

As partições são partes do código, reorganizadas a partir da análise de interdependência de variáveis, para as quais o acoplamento é mínimo e a coesão é máxima. Esta propriedade garante a proximidade com um componente de software.

Como as novas partições redesenham o software original, esta etapa também fornece uma nova hierarquia de chamadas entre as funções do código.

c. Obtenção de Métricas :

Nesta etapa são obtidas métricas relativas à complexidade de cada componente e à flexibilidade para sofrer mudanças [TERNES, 92].

As medidas permitem avaliar o projeto do software, através de critérios que indiquem a necessidade de reimplementação de um componente.

d. Abstração Funcional :

Nesta etapa é gerada uma especificação funcional para cada componente do software, de modo a torná-los formalmente verificáveis [EVANGELISTA, 92].

Esta especificação é baseada em axiomas funcionais para construir expressões algébricas. Tais expressões representam a estrutura funcional do código fonte, que é analisada e convertida

---

2 *Neste trabalho, um subprograma qualquer é considerado uma unidade de programa, pode ser : função, procedimento, subrotina, parágrafos ou seções referenciados por um comando "perform" (linguagem COBOL) e, ainda, cada "entry-point" de uma subrotina ou função (linguagem FORTRAN).*

para uma tabela de decisão funcional.

e. Estruturação de Componentes :

Esta etapa consiste em estruturar os componentes obtidos de acordo com o padrão de especificação adotado e certificá-los. O padrão envolve obter a especificação formal e gerar o desenho a partir da nova hierarquia de componentes. O código pode ser reimplementado, se a implementação existente não satisfizer critérios derivados dos conceitos de abstração de dados e "information hiding"[3]. Finalmente, os componentes certificados são armazenados em um banco de componentes.

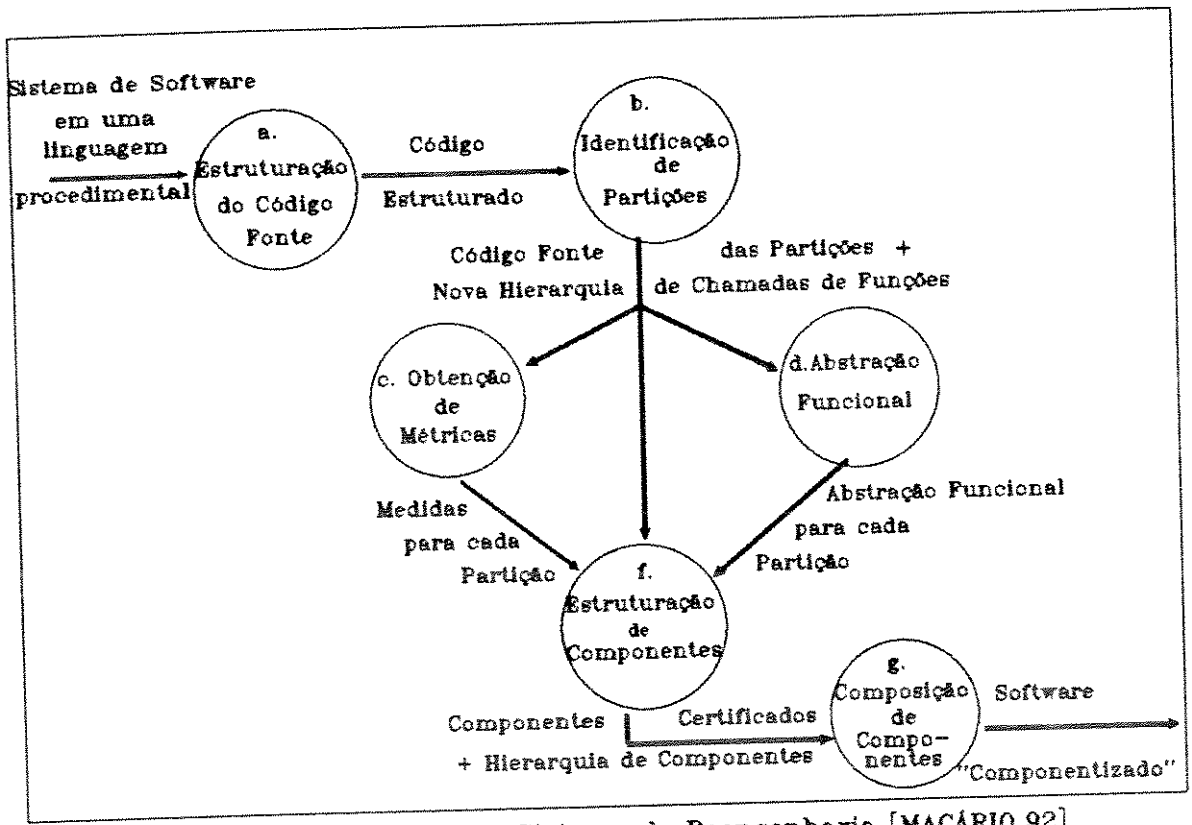

Figura 1.1 : Modelo do Sistema de Reengenharia [MACÁRIO,92]

f. Composição dos Componentes :

Baseando-se no grafo da hierarquia de chamadas os componentes armazenados são reorganizados através de uma linguagem de composição [CAMARGO, 91]. As composições resultantes são validadas contra as suas especificações a partir de uma ferramenta automatizada de certificação de qualidade.

---

3 O princípio de "information hiding" sugere que os módulos (ou componentes) do software podem ser individualmente especificados e projetados de modo que a informação utilizada em cada módulo seja inacessível aos demais [PARNAS, 72].

A **Etapa a**, estruturação do código fonte, tem como objetivo simplificar a estrutura do código original e, ao mesmo tempo, obter uma tradução para um subconjunto da linguagem **C**.

A simplificação da representação do código visa diminuir a complexidade dos algoritmos da **Etapa b** (identificação de partições) e garantir a estrutura de código esperada na **Etapa d** (abstração funcional [EVANGELISTA, 92]). Para isto, o subconjunto da linguagem **C**, adotado para representação do código, é o seguinte :

. ***expressões*** : engloba as definições de expressões para a linguagem **C**, exceto as que incluem chamadas a subprogramas e expressões condicionais,

. ***chamadas a subprogramas*** : somente a referência (chamada) a um subprograma,

. ***"do-while"*** : o único tipo de iteração, e

. ***"if-else"*** : o único tipo de seleção.

A linguagem de representação adotada, subconjunto da linguagem **C**, visa obter a implementação dos componentes em **C**, pois nem toda linguagem de programação permite a abstração de componentes; por exemplo, a linguagem **COBOL** não permite o uso de "information hiding". Além disso, o uso da linguagem **C** facilita a adaptação do produto de software reengenhado a plataformas de desenvolvimento que possuam ambiente UNIX (ou "UNIX-like") [MACARIO, 92].

Assim, o modelo de estruturação utilizado deve construir uma representação padronizada, definida pelo subconjunto da linguagem **C** e dividida em unidades de programa, a partir do código fonte de um produto de software em uma linguagem procedimental não paralela qualquer.

1.2. Revisão Bibliográfica

Encontram-se na literatura várias propostas de padronização de estruturas de controle de programas, com objetivos diversos, que vão da simples estruturação de programas, que visava migrar programas existentes para linguagens de programação estruturada [DJIKSTRA, 79], até a padronização para certificação de qualidade [FENTON, 87] e identificação de paralelismo de execução [AMMARGHELLAT, 92].

A partir da década de 60, o uso de linguagens de programação estruturadas motivou o aparecimento de modelos de estruturação de programas. Jacopini e Bhöm [BHOM, 66] mostraram que é possível representar um sistema a partir de uma máquina de Turing, onde são usadas apenas seleções e seqüências; com esse resultado mostraram que qualquer programa pode ser levado a uma única forma de iteração, denominada "while-form", com o uso de variáveis auxiliares. A "while-form", cuja composição é dada apenas por seqüências, seleções e repetições, é o resultado da normalização do grafo de fluxo de controle de um programa. Normalizar um grafo de fluxo de controle de um programa é transformar o mesmo em um novo grafo com um único ponto

de entrada e um único ponto de saída, composto por seqüências, seleções e repetições. Vários métodos de estruturação foram baseados nesses resultados, com pequenas variações, como os de :

. *Ashcroft e Manna [ASHCROFT, 71]* : o método propõe a transformação de todo "goto" em uma "while-form" com a introdução de variáveis adicionais ligadas a cada predicativo de interesse; isto é, predicativo ao qual um "goto" é associado. O método procura manter uma estrutura de controle próxima à original.

. *Urschler [URSCHLER, 75]* : o método, embora permita o uso de "goto's", leva todos os desvios para trás, minimiza-os, cria vários níveis de "exit loops" e poucas variáveis adicionais. Esta forma de normalização introduz várias iterações no programa, procurando reduzir as replicações de partes do mesmo.

. *Baker [BAKER, 77]* : o método, também baseado em normalização de grafos de fluxo de controle de programas, procura a não duplicação de funções e predicados identificando nós retro-dominadores no grafo do programa original e realizando transformações que os conservem.

. *Knuth [KNUTH, 77]* : na mesma linha da proposta de Ashcroft e Manna, este método introduz variáveis auxiliares a cada transformação, preocupando-se contudo com a eficiência das transformações obtidas.

. *Cowell [COWELL, 79]* : com base na normalização de grafos de fluxo de controle de programas, propõe uma teoria sobre programas abstratos para sintetizar estruturas de controle, facilitando a análise e estruturação de programas.

. *Linger e Mills [LINGER, 79]* : este método divide-se em duas fases, das quais a primeira é um simples seqüenciamento do programa, transformando o mesmo em uma "while-form" controlada por uma única variável auxiliar, que funcionalmente equivale a um contador de instruções. A segunda parte faz a composição das seqüências obtidas, a partir da análise semântica do contador adicional e reduzindo o uso do mesmo; porém, para obter esse resultado o método permite a replicação de partes do programa no programa estruturado final.

. *Gattaz [GATTAZ, 82]* : propôs o uso do método de Linger e Mills para código fonte em COBOL, a partir do seqüenciamento de cada linha de comando de um programa. No entanto, esse trabalho não propõe uma solução adequada para o emaranhamento de unidades de programa ocasionado por desvios e sobreposições funcionais. Este tipo de emaranhamento é muito comum em programas COBOL; ocorre devido ao fato da linguagem permitir que partes do programa sejam utilizadas como subprogramas e, ao mesmo tempo, como um conjunto de comandos executados seqüencialmente.

. *Oulsnam ([OULSNAM, 82] e [OULSNAM, 87])* : esse método de estruturação identifica formas básicas não estruturadas transformando-as em formas estruturadas; o método é baseado na identificação de nós retro-dominadores do grafo de fluxo de controle [BAKER, 77]. Cada transformação leva à inclusão de uma nova variável; a variável corresponde a um predicativo para o qual existe um desvio associado. O número de transformações é minimizado em relação aos outros métodos; permite a escolha entre a inclusão de novas variáveis ou a replicação de partes do programa no programa estruturado final. Tse [TSE, 87] propôs novas simplificações para o método de Oulsnam

que minimizam o número de variáveis criadas.

. *Ammarghellat [AMMARGHELLAT, 92]* : o método de normalização do grafo de fluxo de controle objetiva identificar paralelismos e é extremamente simples; baseia-se em relações de dependências funcionais, que são levantadas pelo método e obtidas de forma semelhante ao seqüenciamento e composições propostos por Linger e Mills. O método reduz o número de construções sintáticas, porém permite que sejam utilizados "goto's" para criar ciclos no fluxo de controle.

A partir do trabalho de McCabe [McCABE, 76], onde foi proposta uma métrica de complexidade baseada no número de ciclos de um programa, outros trabalhos mostraram que a complexidade de programas não estruturados não é adequadamente representada por essa métrica ([ELSHOFF, 78] e [OULSNAM, 79]). Novos trabalhos, envolvendo esta e outras propostas de métricas de complexidade de programas, buscaram a padronização de estruturas de controle, para viabilizar o processo de cálculo de métricas, como os de :

. *Fenton ([FENTON, 86] e [FENTON, 87])* : mostrou que é imprescindível decompor o programa em programas primos[4], para obter uma forma padronizada, que viabiliza o cálculo de métricas de complexidade estrutural sobre um programa; onde, a decomposição é realizada sobre o grafo de fluxo de controle do programa.

. *Gannon [GANNON, 83]* : propõe um método de decomposição do grafo de programa em programas primos, estruturando-o de modo a obter uma base adequada para calcular métricas estruturais.

. *Bail [BAIL, 88]* : decompõe os programas analisados em programas primos em sua forma mais simples; mede complexidade sobre número de primos obtidos, variáveis e operadores em cada primo.

. *Cantone [CANTONE, 86]* : obtém métricas de complexidade sobre programas estruturados; converte formas não estruturadas em formas estruturadas, procurando caminhos que contenham programas próprios[5] no grafo de fluxo de controle do programa original.

Os métodos baseados em grafos de fluxo de controle de programas podem ser automatizados visando a estruturação e obtendo a padronização esperada. No entanto, é necessário que se disponha de uma ferramenta genérica para a geração automática de grafos de programa; isto é, o grafo de fluxo de controle do programa possa ser obtido pela ferramenta qualquer que seja a linguagem de programação original.

Carnassale [CARNASSALE, 91] propôs e desenvolveu uma ferramenta multi-linguagem para obter grafo de fluxo de controle de programas. O grafo é gerado a partir de uma linguagem intermediária, denominada LI. A LI não é uma linguagem que possa ser compilada e executada, pois é apenas um mapeamento para as estruturas de controle do código fonte original. Se um modelo de estruturação for implementado sobre o grafo

---

4 Programas primos são unidades mínimas e básicas para a construção de programas estruturados.

5 Programas próprios são subdivisões do grafo de fluxo de controle, para as quais existe apenas um único ponto de entrada e um único ponto de saída e para todo nó existe um caminho do ponto de entrada ao nó e do nó ao ponto de saída.

de fluxo de controle obtido para um programa qualquer, a representação do programa estruturado final precisaria ser feita na linguagem de programação original.

Estudos ou métodos de estruturação de programas com base em uma única linguagem de programação, criam dependências em relação às estruturas de controle permitidas pela linguagem, o que nem sempre viabiliza a sistematização e conseqüente implementação do mesmo. Por exemplo :

. *Ramshaw [RAMSHAW, 88]* : propôs um método de estruturação para programas em FORTRAN, que leva a outro programa também em FORTRAN, mas preocupando-se em manter a estrutura do programa original. Aparentemente, o método levaria a uma estrutura cognitivamente melhor, mas é de difícil implementação, devido à complexidade das análises realizadas sobre o código original e às características da linguagem.

. *Ferramentas automáticas de estruturação* : de código fonte COBOL para código fonte COBOL apresentam o mesmo problema. Alguns exemplos são as ferramentas IBM COBOL Structuring Facility ([LINGER, 88] e [HAUSLER, 90]), Retrofit [MILLER, 90], Superstructure [MORGAN, 84] e o RECORDER [BUSH, 85]. Nem sempre a linguagem COBOL permite representar, ou mesmo suporta, estruturas de controle adequadas a um programa estruturado devido às suas características; vários trabalhos na literatura mostram essas inadequações ([MILLER, 87], [CALLIS, 88] e [MILLER, 90]).

Logo, um método de estruturação não pode ser genérico sobre o grafo de fluxo de controle do programa; é necessário considerar a linguagem utilizada na implementação do programa original e suas limitações. Uma alternativa seria o uso de tradutores fonte a fonte; o programa original seria traduzido e o grafo de fluxo de controle do programa traduzido seria a base do modelo de estruturação.

Métodos ou ferramentas que fazem traduções fonte a fonte, em sua maioria, convertem estruturas de uma linguagem em estruturas equivalentes em outra linguagem, não modificando a estrutura original. Se o sistema original não é bem estruturado, em termos de arquitetura e fluxo de controle, então o produto final da tradução continua apresentando um nível de qualidade pobre e possuindo código ineficiente e de difícil manutenção [BYRNE, 91].

A estrutura gerada por um tradutor não seria uma limitação, uma vez que o interesse é estruturar o resultado obtido; no entanto não são disponíveis tradutores fonte a fonte de várias linguagens procedimentais para a linguagem de programação **C**, que é a linguagem alvo neste trabalho. Por exemplo, encontra-se tradutor **FORTRAN** para **C**, de domínio público [FELDMAN, 90]; porém não existe a mesma solução para a linguagem **COBOL**.

1.3 Objetivo e Organização do Trabalho

O **Modelo de Estruturação** proposto é uma adaptação do método de Mills e Linger [LINGER, 79]. Para o modelo proposto o código fonte de um produto de software, implementado em uma linguagem procedimental

não paralela, será representado por um sub-conjunto da linguagem de programação **C**.

A representação do código em um sub-conjunto de **C** é composta por unidades de programa contendo três tipos de estruturas de controle : comandos seqüenciais (expressões e chamadas a subprogramas), iterações de um único tipo ("do-while") e seleções simples ("if-else").

O modelo envolve a transformação de estruturas de controle da linguagem de programação original; assim, existe uma dependência em relação à linguagem de programação do código fonte original.

Foram selecionadas três linguagens de programação para este trabalho : **COBOL**, **FORTRAN** e **C**. A linguagem **COBOL** por ser amplamente utilizada e possuir um alto custo de manutenção [YEH, 90-a]; além disso, programas em linguagem **COBOL** são um excelente exemplo de aplicação para o **Modelo de Estruturação**, pois a linguagem apresenta estruturas de controle que permitem a construção de códigos "spaghetti-like" e, conseqüentemente, que dificultam a implementação de processos automáticos de estruturação ([MILLER, 91], [MILLER, 87] e [HAUSLER, 90]). A linguagem **FORTRAN** por ser uma das mais exploradas e complexas para obtenção de métodos automáticos de estruturação e tradutores fonte a fonte ([RAMSHAW, 88], [GANNON, 83], [BYRNE, 91] e [FELDMAN, 90]). E, a linguagem **C** por ser a escolhida para a representação dos resultados do **Modelo de Estruturação**.

No Capítulo 2 são apresentados os conceitos básicos relativos ao método de Mills e Linger, transformações de estruturas de controle genéricas e o **Modelo de Estruturação**.

O **Modelo de Estruturação** é dividido em duas etapas, onde a primeira depende da linguagem de programação original e a segunda depende apenas do resultado obtido na primeira.

A dependência da linguagem de programação leva à necessidade de modelar a primeira etapa para cada linguagem; por isso, são propostos modelos para as linguagens **COBOL**, **FORTRAN** e **C**, separadamente, no Capítulo 3. A segunda etapa do **Modelo de Estruturação**, que depende apenas do resultado da primeira, é dada no Capítulo 4.

No Capítulo 5 encontra-se uma proposta para implementação do modelo, sob ambiente UNIX, e a descrição dos protótipos implementados. Ainda, no Capítulo 5, é apresentado um exemplo da aplicação dos protótipos, sobre o código fonte de um produto de software, e uma análise do resultado obtido.

O Capítulo 6 contém as considerações finais do trabalho, incluindo as contribuições do modelo e discussões sobre trabalhos futuros.

Os apêndices A, B e C correspondem, respectivamente, às transformações de cada estrutura de controle das linguagens **COBOL**, **FORTRAN** e **C** para o padrão de estruturação adotado. O apêndice D contém as complementações dos exemplos dados no Capítulo 5; dados utilizados para o teste do produto de software estruturado e código fonte de algumas unidades de programa do código fonte do produto de software antes e após estruturação.

2. DEFINIÇOES BASICAS E O MODELO DE ESTRUTURAÇAO

O **Modelo de Estruturação** apresentado neste trabalho é uma adaptação da proposta de Linger e Mills para a estruturação de programas ([LINGER, 79]).

O modelo consiste em obter uma representação padronizada do código fonte original, implementado em uma linguagem de programação procedimental não paralela qualquer, em **unidades de programa** representadas pelo sub-conjunto da linguagem de programação **C**.

Com base na normalização da estrutura de controle de um programa da proposta de Linger e Mills, as transformações para estruturas de controle presentes em linguagens de programação procedimentais não paralelas são generalizadas; o **Modelo de Estruturação** é especificado por algoritmos aplicados sobre um tipo abstrato de dados usado para representar uma **unidade de programa**.

Os conceitos e definições que formam a base do **Modelo de Estruturação** são apresentados a seguir.

2.1. Definições Básicas

2.1.1. Programas Próprios e Programas Primos

Um **programa** pode ser visto como uma função que mapeia o conjunto de dados de entrada para o conjunto de dados de saída, através de uma **estrutura de controle** [LINGER, 79].

A **estrutura de controle** pode ser representada por um grafo dirigido composto por nós funcionais, nós predicativos e nós coletores [LINGER, 79]. Sua função é apenas preservar a ordem em que as funções e predicados ocorrem, ignorando a identidade dos mesmos.

Um **nó funcional** tem apenas um arco de entrada e um arco de saída. Seu efeito sobre os dados pode ser representado por uma função matemática, como em expressões do tipo *"a=b+c"*, *"i=log(sqrt(n))"*, ou *"a=imprime(matriz,n,m)"*. Um nó funcional pode ser representado por um grafo dirigido como ilustrado na Figura 2.1.

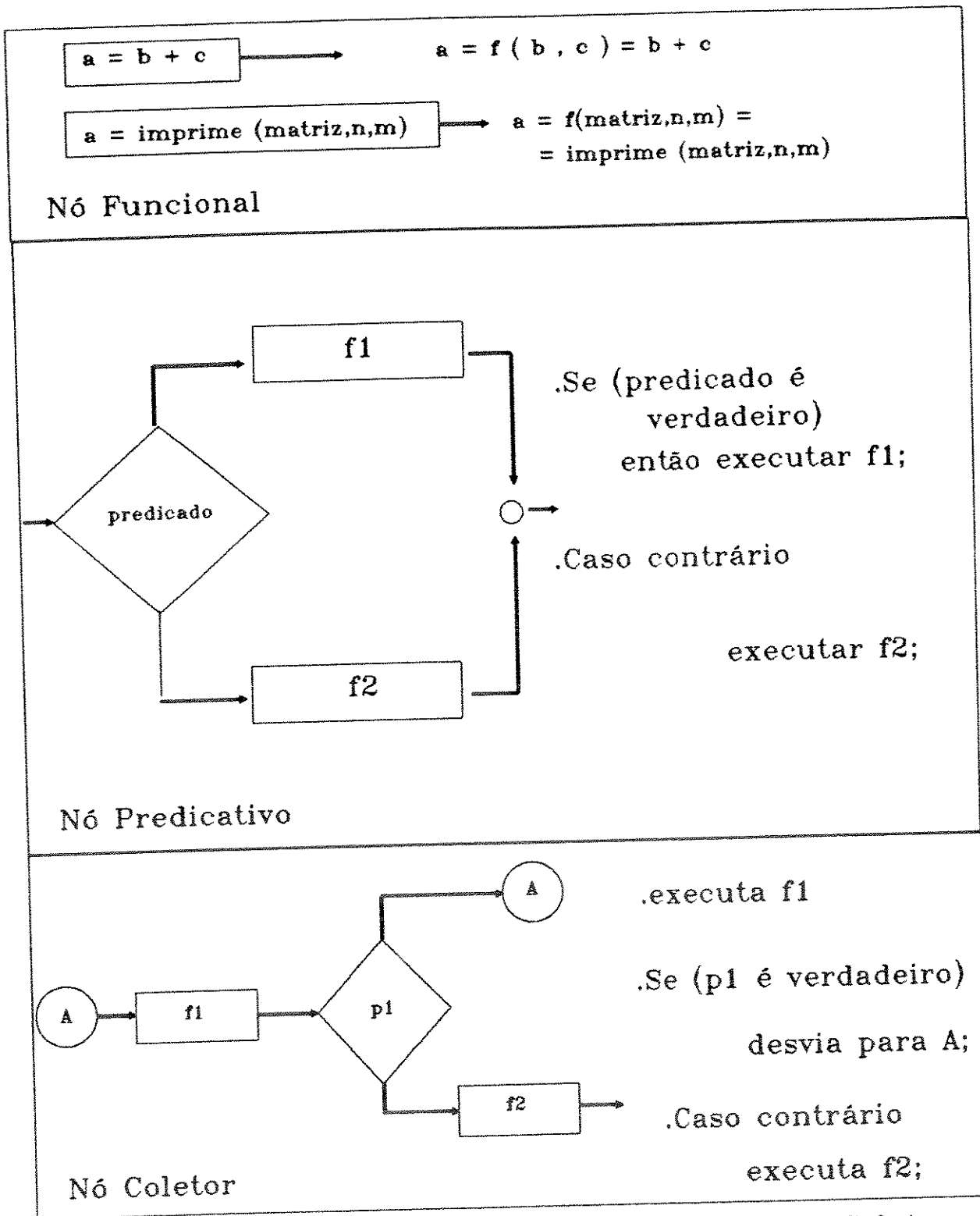

Figura 2.1: Nó Funcional, Nó Predicativo e Nó Coletor

Um **nó predicativo** tem um arco de entrada e um único arco de saída, existindo dois caminhos possíveis para o arco de saída; a escolha do caminho de saída depende do valor do predicado associado ao nó. Qualquer comando de decisão é um nó predicativo; por exemplo, *"if"* de uma linguagem de programação - **C**, **PASCAL**, **COBOL**, etc; como ilustrado na Figura 2.1.

Um **nó coletor** não realiza nenhuma operação, consiste de mais de um arco de entrada e apenas um arco de saída. Sua funcionalidade é a de um rótulo de instrução; portanto, qualquer representação de rótulo de instrução é um nó coletor. Por exemplo, um desvio para uma instrução em uma linguagem de programação procedimental qualquer envolve um comando e um rótulo, como *"goto a;"*, na linguagem **C**. No exemplo, *"a"* é um rótulo, que pode ser referenciado mais de uma vez. A Figura 2.1 ilustra um nó coletor associado ao rótulo *"a"*.

Um programa cuja estrutura de controle tem apenas um arco de entrada e um arco de saída, e para cada um de seus nós possui um caminho do arco de entrada do programa ao nó e do nó ao arco de saída do programa é dito um **programa próprio** [BAIL, 88]. A Figura 2.2 mostra alguns exemplos e contra-exemplos de **programas próprios**.

Todo **programa próprio** pode ser abstraído para um nó funcional e, inversamente, todo nó funcional pode ser expandido em um **programa próprio**, sem que isto afete outras partes do programa. Logo, subpartes do programa com características de **programa próprio** são **sub-programas próprios**.

Um **programa próprio** que não possui **sub-programas próprios** de mais de um nó é um **programa primo**. Embora o número de **programas primos** seja infinito, casos particulares contendo até três nós são as formas usuais da programação estruturada [BAIL, 88], como "**if**", "**while**", "**repeat**", etc. A Figura 2.3 apresenta alguns **programas primos** de até três nós.

Pode-se compor programas trocando-se o nó funcional de um **programa primo** por um outro **programa primo**, o que resulta em um novo **programa próprio**. Um **programa composto** é um **programa próprio** de tamanho arbitrário, obtido pela composição de **programas primos** [LINGER, 79]. A complexidade estrutural de um programa composto pode ser limitada se for fixada uma base de **programas primos** para as composições.

A partir de uma base qualquer de **programas primos**, desde que envolva, no mínimo, a representação de nós funcionais e nós predicativos [BOHM, 66], podem ser geradas todas as combinações de **programas próprios**. Um **programa estruturado** é um **programa próprio**, composto sobre uma base fixa de **programas primos** [LINGER, 79].

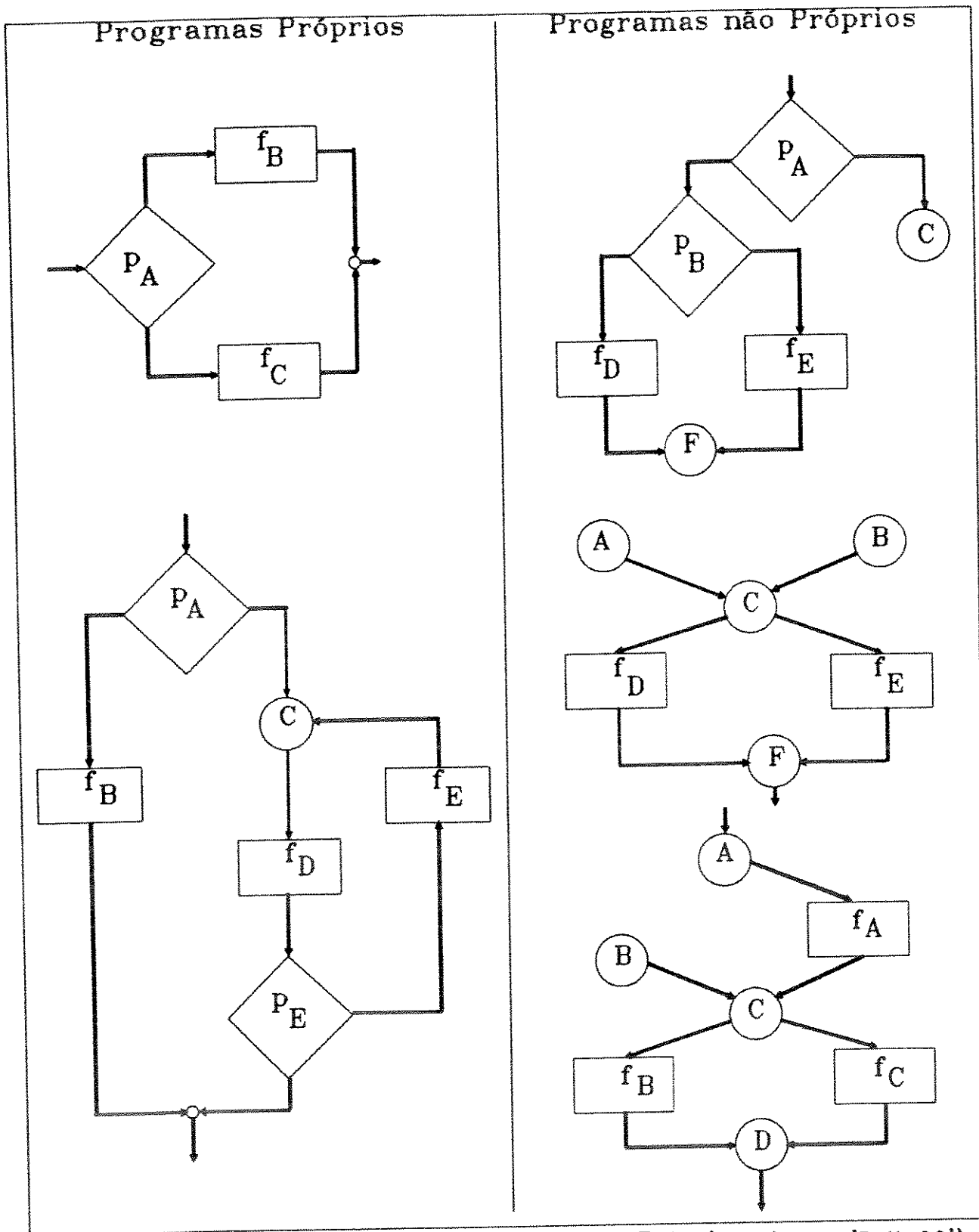

Figura 2.2: Programas Próprios e Não Próprios (Fonte [Bail, 88])

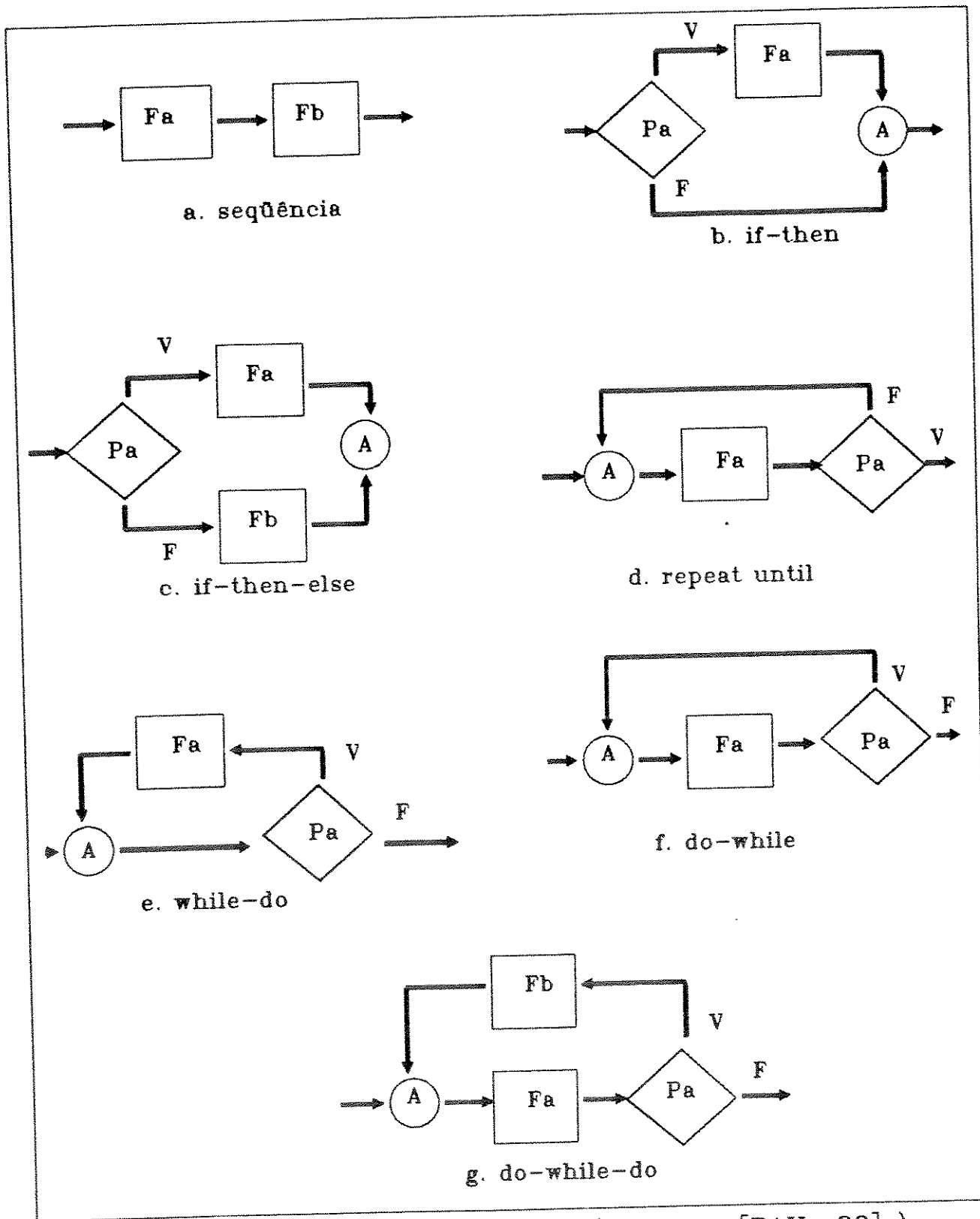

Figura 2.3 : Programas Primos (Fonte : [BAIL, 88])

2.1.2. Teorema da Estrutura e Primeira Forma Normal

As propriedades e conceitos anteriores levam ao seguinte teorema:

Teorema da Estrutura :

> "Qualquer **programa próprio** é funcionalmente eqüivalente a um programa estruturado, construído a partir de uma base fixa de **programas primos** { seqüência, seleção simples, repetição}, usando as funções e predicados do programa original e atribuições e testes sobre um contador adicional." [LINGER, 79]

Antes de ilustrar a aplicação do teorema, precisamos conceituar o que chamamos de eqüivalência funcional.

Quando falamos em **eqüivalência funcional** estamos pressupondo que os dois programas têm a mesma função, isto é, realizam as mesmas transformações sobre um conjunto de dados. Ou ainda, segundo Marcotty [MARCOTTY, 75], dois programas são funcionalmente eqüivalentes se produzem o mesmo resultado para dados de entrada idênticos. Isto não significa que os dois programas tenham eqüivalência de execução. Eqüivalência de execução implica que as transformações sobre os dados sejam implementadas da mesma forma [LINGER, 79].

Mostraremos a aplicação do teorema, considerando um **programa próprio** qualquer com um número arbitrário de funções e predicados, cujo ponto de entrada é o primeiro comando executável do programa e o ponto de saída é o último comando executável.

Tomemos, como exemplo, o fluxograma da Figura 2.4, que é um **programa próprio**, pois apresenta apenas um ponto de entrada e um ponto de saída e para todo nó existe um caminho do ponto de entrada ao ponto de saída do programa passando pelo nó; no entanto, não é um programa estruturado, existem desvios para blocos internos às decisões, isto é, nem todos os seus sub-programas são **próprios**.

O fluxograma da Figura 2.4 corresponde ao seguinte programa [GANNON, 83], originalmente em **FORTRAN** e aqui convertido para a linguagem **C** :

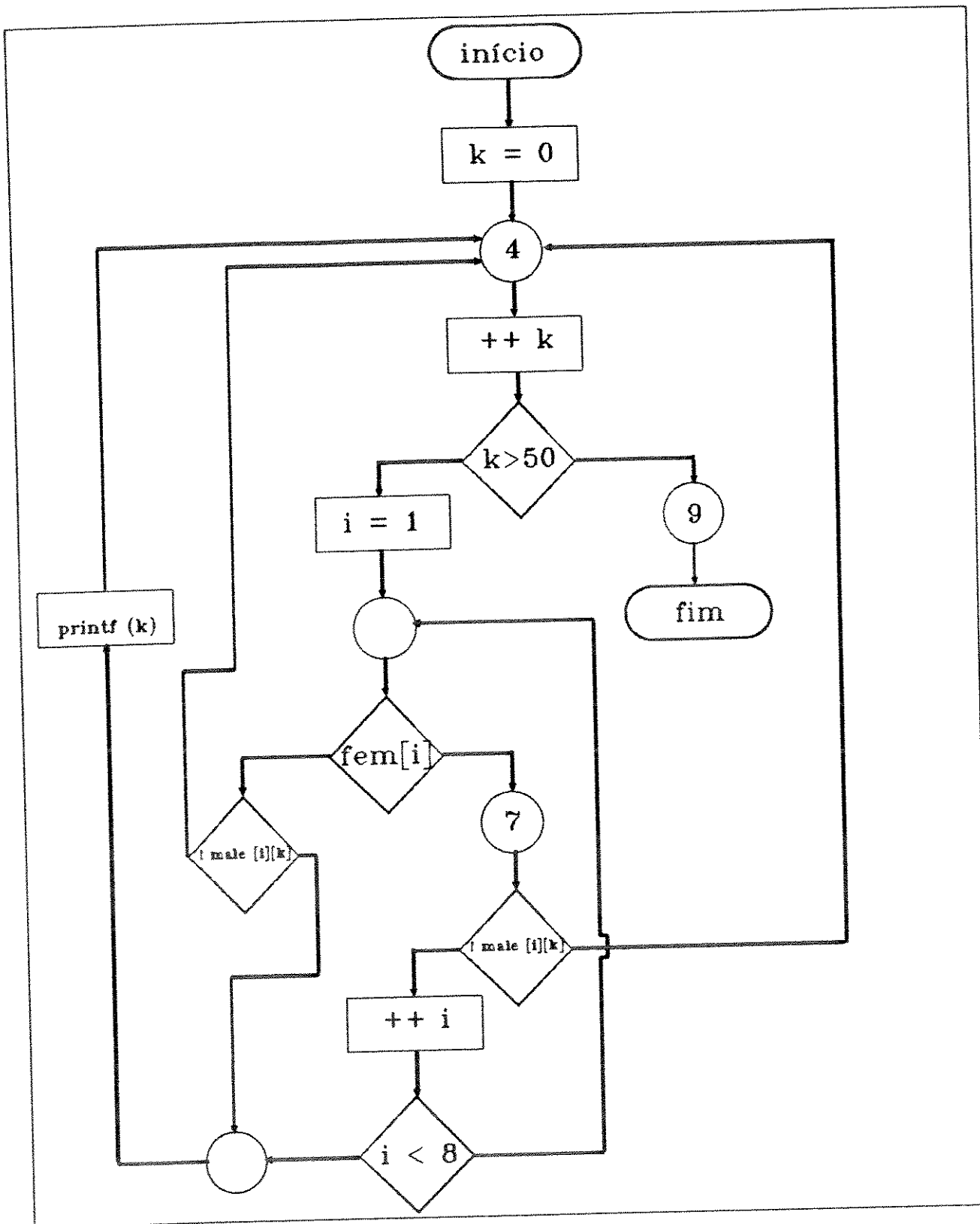

Figura 2.4 : Fluxograma Exemplo

```
void        exemplo(fem,male)
int         fem[50],male[8][50];
{
            int k,i;
            k = 0;
quatro:     ++k;
            if ( k > 50 )
                        goto nove;
            i = 1;
            do {
                  if ( fem[i] )
                        goto sete;
                  if ( ! male[i][k] )
                        break;
                  goto quatro;
sete :            if ( ! male[i][k] )
                        goto quatro;
                  ++i;
            } while ( i < 8 );
            printf ("k=[%d] \n", k );
            goto quatro;
nove :      ; /* nada */
}
```

Se todos os comandos do programa forem numerados, sendo o inicial de *número 1* e o final do programa o de *número 0*, e a cada comando for acoplado o número de seu sucessor, através do uso de um contador auxiliar *"l"*, cada comando será transformado em *seqüências* do seguinte tipo :

. para todo nó funcional f_i :

$$i : f_i,\ l=j$$

. para todo nó predicativo p_i :

$$i : \text{if } (p_i) \quad \text{then } l = j \quad \text{else } l = n$$

Onde,

i : número do comando associado à *seqüência*

j : número do comando / *seqüência* sucessora. Para o nó predicativo, número do comando / *seqüência* sucessora caso o predicativo seja verdadeiro.

n : para o nó predicativo, número do comando ou *seqüência* sucessora quando o predicativo é falso.

l : contador auxiliar

Deve-se notar que a representação de cada comando como uma *seqüência* não altera a sua funcionalidade; a variável auxiliar *"l"* introduzida é apenas um contador de instruções, isto é, indica qual

será a próxima ***seqüência*** a ser executada.

Comandos que provocam desvios podem ser representados apenas pelo seu número de ***seqüência*** e a indicação da ***seqüência*** sucessora, que corresponde ao ***"rótulo"*** (nó coletor) para onde o desvio é dirigido [GATTAZ, 82], da seguinte forma :

$$i : l = j$$

Voltando ao exemplo da Figura 2.4, seqüenciam-se todos os comandos executáveis; deve-se lembrar que o fim do programa é representado por zero, mas de fato não existe uma ***seqüência*** com esta numeração. Assim, as ***seqüências*** que compõem o programa serão formadas como ilustrado na Figura 2.5.

Para reorganizar as ***seqüências***, respeitando sua estrutura de controle original, deve-se construir dois **programas primos**. O primeiro é um nó funcional simples, que inicia o contador auxiliar e corresponde à atribuição ***"l=1"***. O segundo um ***"do-while"*** (vide Figura 2.3), cujo bloco interno são as ***seqüências*** encontradas.

E, ainda, a cada ***seqüência*** é acoplado o teste do valor do contador adicional, como para as duas ***seqüências*** seguintes :

. i-ésima ***seqüência*** com nó funcional f_i :

```
if ( l==i ) {
        fi;
        l=j;
}
```

. k-ésima ***seqüência*** com nó predicativo p_k :

```
if ( l==k ) {
     if ( pk ) {
          l = j;
     }
     else {
          l = n;
     }
}
```

O ***"do-while"*** terminará sua execução quando tiver ocorrido um desvio para o fim do programa, que será representado pela existência de uma ***seqüência*** sucessora de número zero, ou seja, um desvio para fora do programa é ***"i : l = 0"***.

A Figura 2.6 mostra a composição das ***seqüências*** (ilustradas na Figura 2.5) através do comando inicial ***"l=1"*** e do ***"do-while"*** que controla a execução, que é a aplicação do ***Teorema da Estrutura*** ao programa do exemplo; ou por Linger e Mills [LINGER, 79], a prova do teorema.

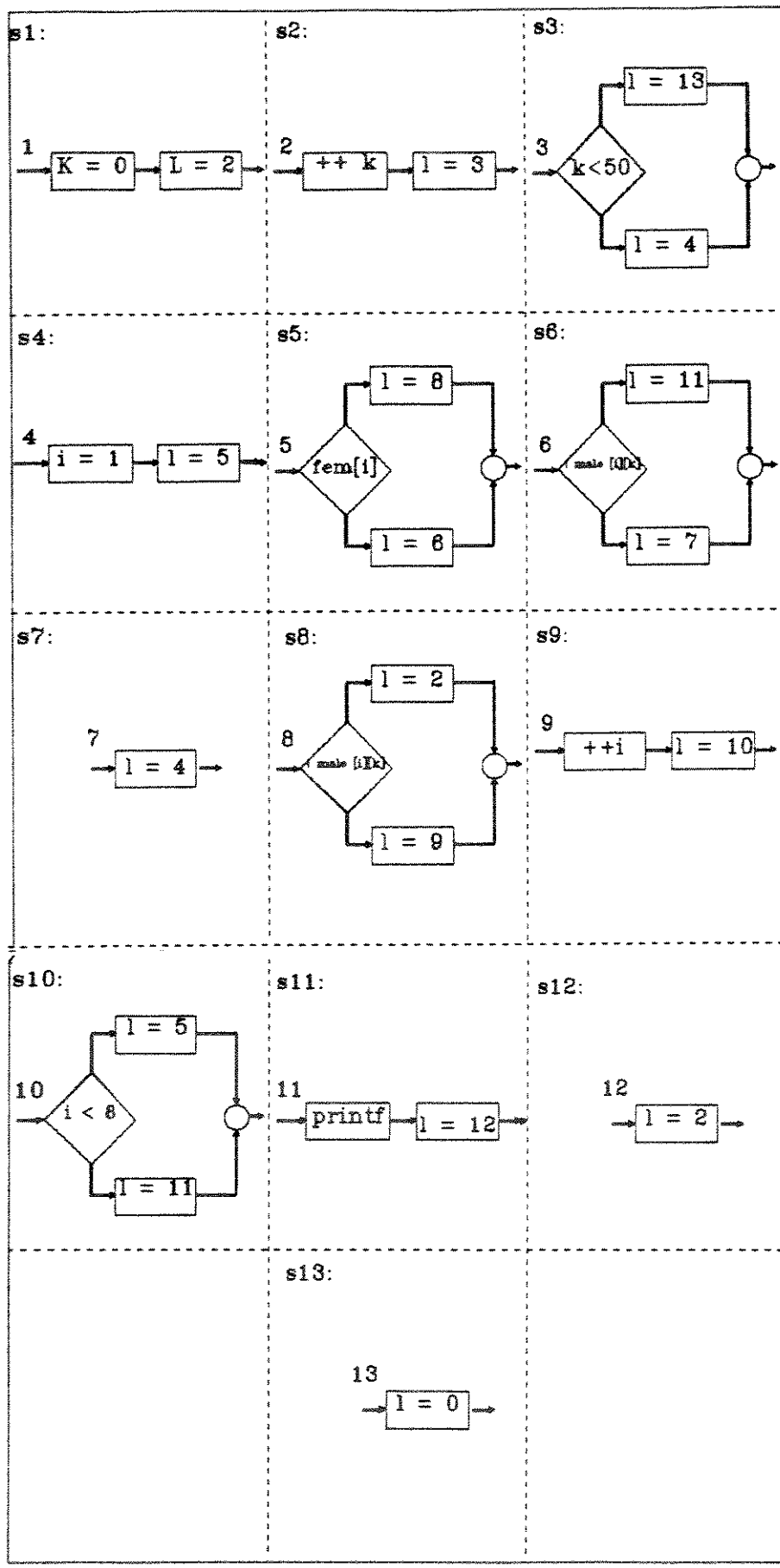

Figura 2.5 : Seqüências para Fluxograma Exemplo

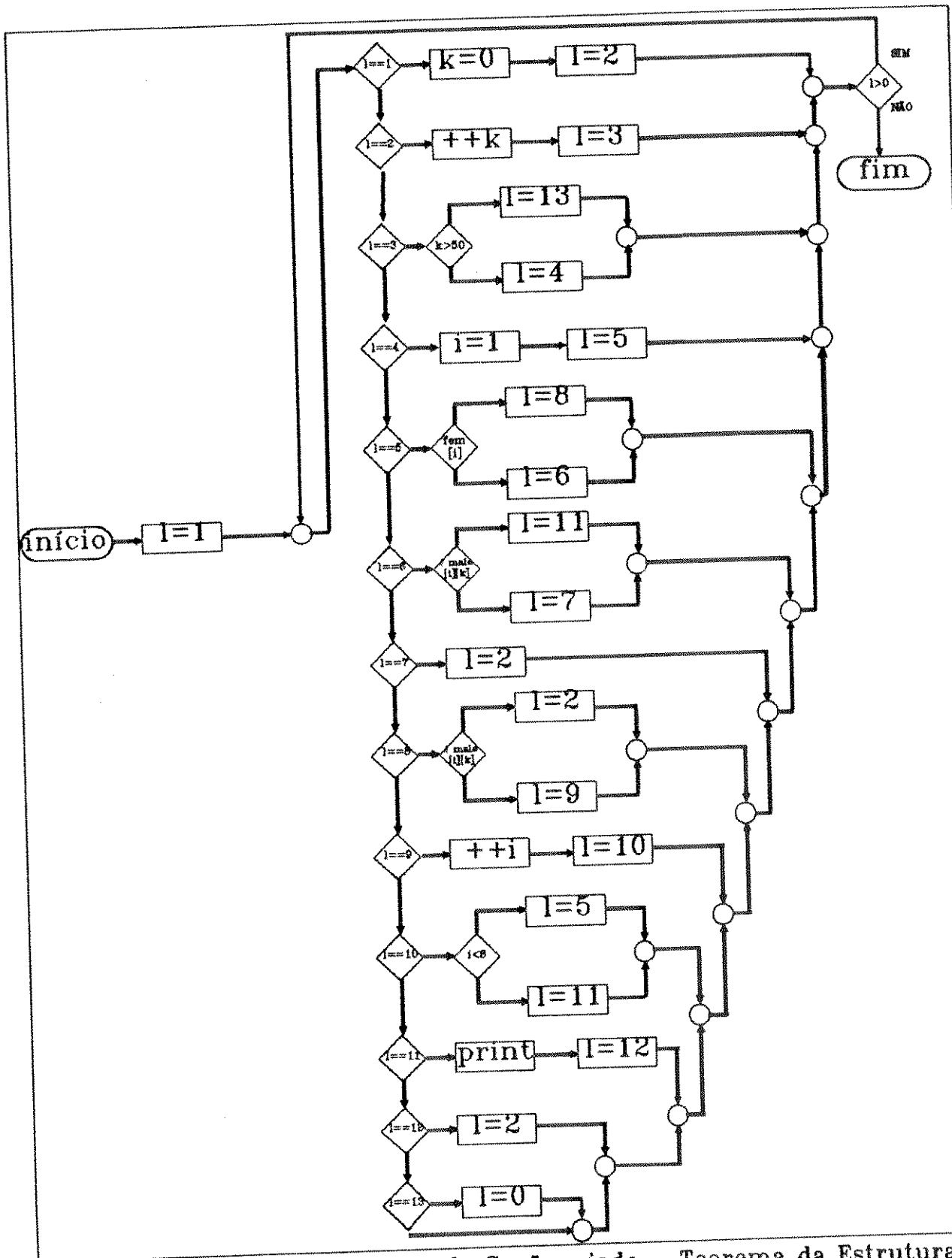

Figura 2.6 : Fluxograma Exemplo Seqüenciado – Teorema da Estrutura

Deve-se notar que o programa da Figura 2.6 está estruturado, é um **programa próprio** composto por **programas primos** e a ordem de execução das ***seqüências*** foi mantida devido ao uso do contador *"1"*. O uso do contador adicional *"1"* não causa nenhum efeito colateral sobre o programa estruturado, sua função é manter a equivalência funcional entre o programa estruturado e o original.

Todo programa obtido a partir da aplicação do **Teorema da Estrutura** encontra-se em uma forma padronizada, que é uma decomposição em **programas primos** a partir de uma base fixa, mínima, necessária e suficiente de **programas primos** [LINGER, 79]. A esta forma damos o nome de **Primeira Forma Normal** [GATTAZ, 82].

Esta forma de representação de um programa é básica para qualquer transformação e/ou operação automatizada subseqüente; por exemplo, a reorganização de blocos funcionais a partir de novas composições de **programas primos**.

Além disso, como a estrutura obtida particiona cada bloco funcional do código em ***seqüências*** básicas e mínimas, o número de ***seqüências*** obtidas é uma medida eficaz e completa do número de linhas de código executáveis do programa, pois cada seqüência corresponde a um nó funcional ou predicativo do código original. A medida é considerada eficaz e completa porque pode ser obtida a partir de uma padronização normalizada do código [FENTON, 87].

2.1.3. Simplificação da Primeira Forma Normal

Embora a **Primeira Forma Normal** seja estruturada, ela não possui clareza e eficiência suficientes. Para melhorá-la é obtida uma nova construção, eliminando atribuições e testes desnecessários do contador adicional *"1"* [LINGER, 79].

A idéia é substituir, para algum *"1>0"*, toda atribuição *"1=j"* pela **seqüência** s_j [LINGER, 79]. Ou seja, compõem-se **programas primos** substituindo um nó funcional do tipo *"1=j"* pelo **programa primo** correspondente à **seqüência** s_j, o que resulta em um novo **programa próprio** (propriedade de composição de **programas primos** [LINGER,79]). E, se para um determinado valor de *"j"* não existir uma atribuição correspondente do tipo *"1=j"*, então a seqüência s_j pode ser removida do programa.

A única barreira para continuar o processo de substituição é a existência de uma referência recursiva a alguma **seqüência**, o que corresponde à atribuição *"1=j"* em uma **seqüência** s_j [LINGER,79]. Em tal caso, substituir *"1=j"* por s_j, reintroduziria a atribuição *"1=j"* e, assim, recursivamente em um processo infinito.

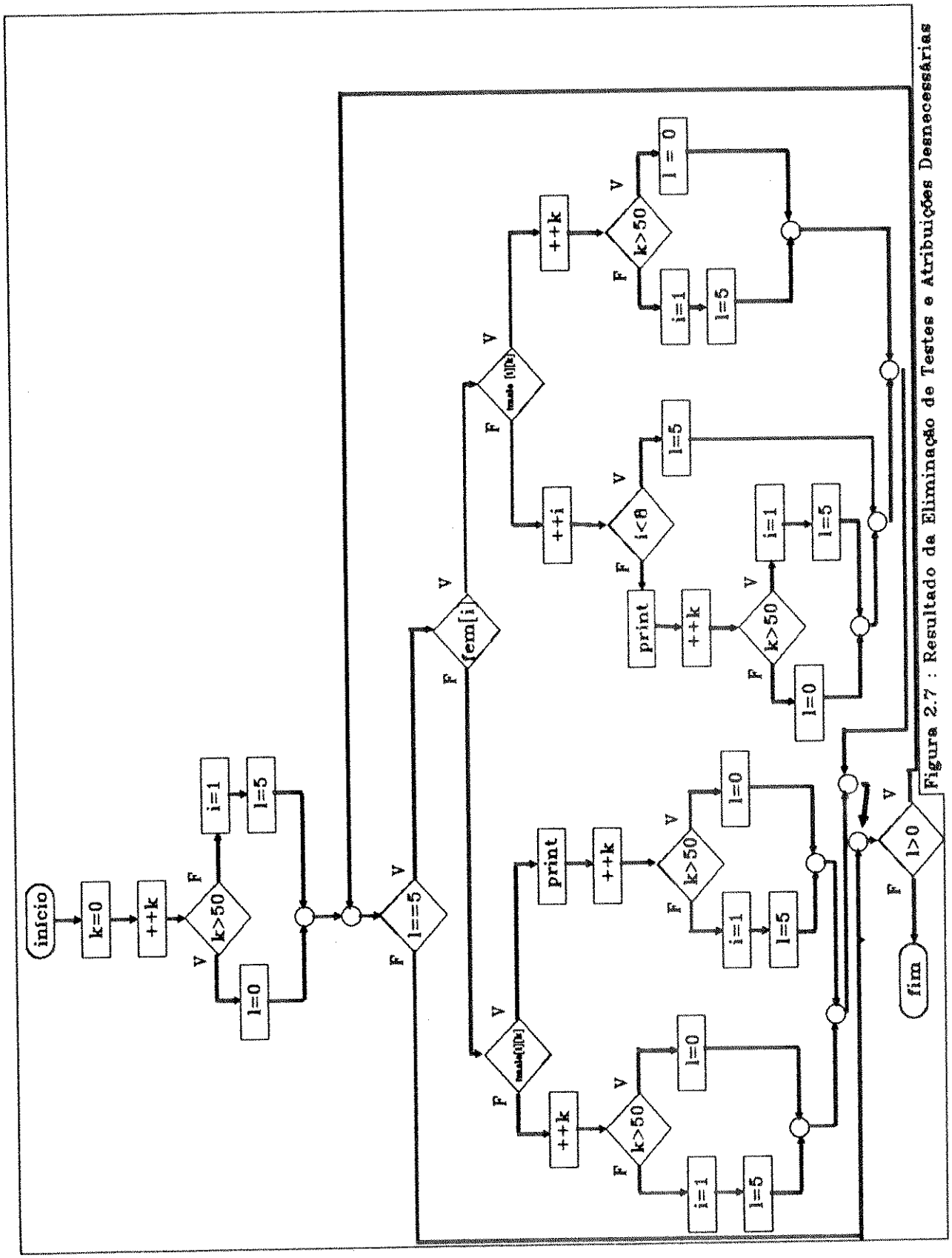

Figura 2.7 : Resultado da Eliminação de Testes e Atribuições Desnecessárias

Assim, as substituições são realizadas até que :

. todas as atribuições a "l" tenham sido eliminadas, exceto "l=0", ou

. toda seqüência s_j remanescente, j>0, possua uma atribuição "l=j".

Utilizando essas composições sobre a **Primeira Forma Normal** ilustrada na Figura 2.6 é obtido o resultado ilustrado na Figura 2.7.

Note-se que o resultado da Figura 2.7 possui uma série de replicações de parte do programa original; existem tantas re-ocorrências de uma determinada **seqüência** s_1, da **Primeira Forma Normal**, quantos forem as atribuições *"l=i"* em **seqüências** s_j, *i diferente de j*. Também deve-se notar que, por ser uma **seqüência** recursiva, s_6 é a única que não foi eliminada.

2.1.4. Estruturas de Controle e Seqüenciamento

Para decompor um programa em **programas primos** utiliza-se o **Teorema da Estrutura**. O teorema garante a obtenção de uma forma estruturada de um **programa próprio** a partir de uma base fixa de **programas primos**.

Pela propriedade de composição de **programas primos** pode-se gerar todas as combinações possíveis de **programas próprios**. Logo, qualquer estrutura de controle que seja um **programa próprio** é representável por uma composição de **programas primos**.

Segundo Pratt [PRATT, 84], as **estruturas de controle**, vistas de um nível de abstração genérico, podem ser resumidas em :

1. **Composição** : um ou mais comandos colocados em uma seqüência textual em um programa, ou ainda, um ou mais nós funcionais executados em ordem seqüencial.

2. **Seleção** : duas ou mais composições de comandos de um programa podem formar alternativas, tal que apenas uma das composições seja executada.

3. **Iteração** : representa uma composição de comandos, que pode ser executada repetidamente zero ou mais vezes.

4. **Desvios** : um desvio transfere o controle de execução de um nó funcional, em uma composição, para um outro nó funcional (e/ou composição) rotulado.

5. **Subprogramas** : chamadas de programas que são executadas como um nó funcional simples, isto é, transferem o fluxo de execução para uma composição no programa que, quando termina, devolve o controle de execução ao seu caminho (fluxo) original.

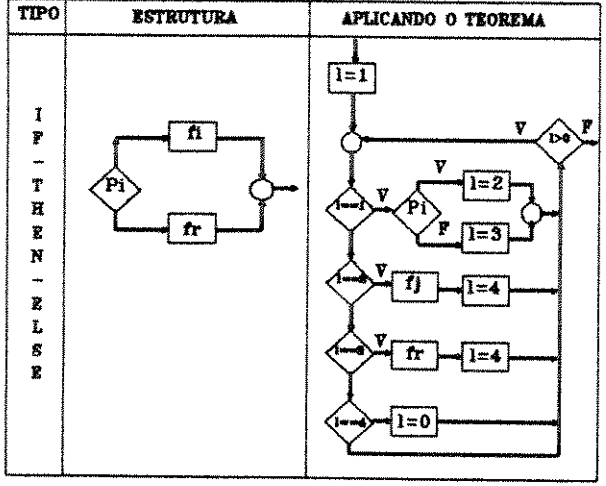

Tabela 2.1 : Teorema da Estrutura
Aplicado às Estruturas de Controle – Parte b

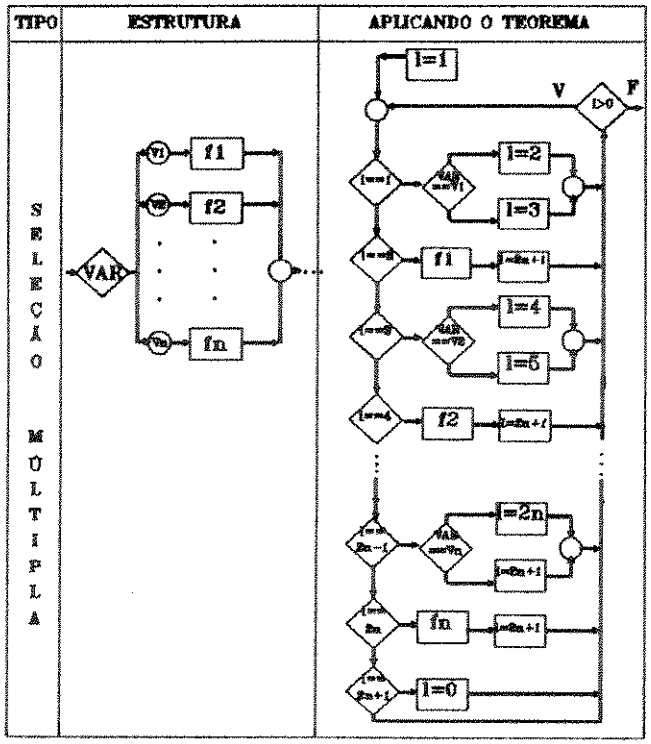

Tabela 2.1 : Teorema da Estrutura
Aplicado às Estruturas de Controle – Parte c

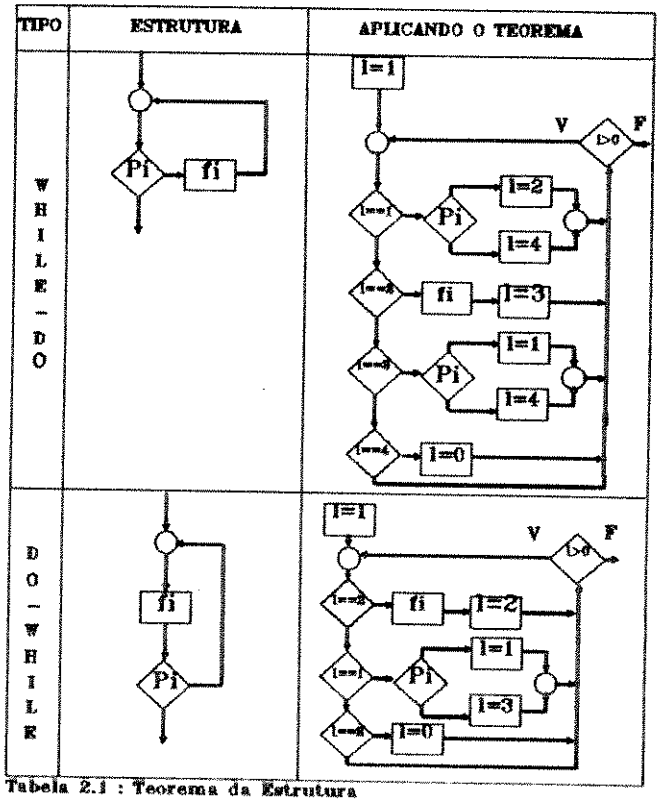

Tabela 2.1 : Teorema da Estrutura
Aplicado às Estruturas de Controle – Parte d

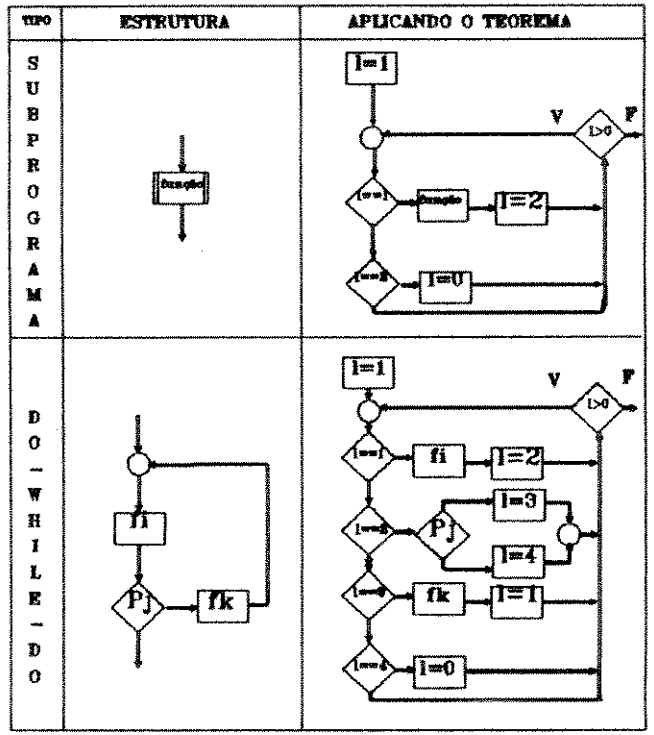

Tabela 2.1 : Teorema da Estrutura
Aplicado às Estruturas de Controle – Parte e

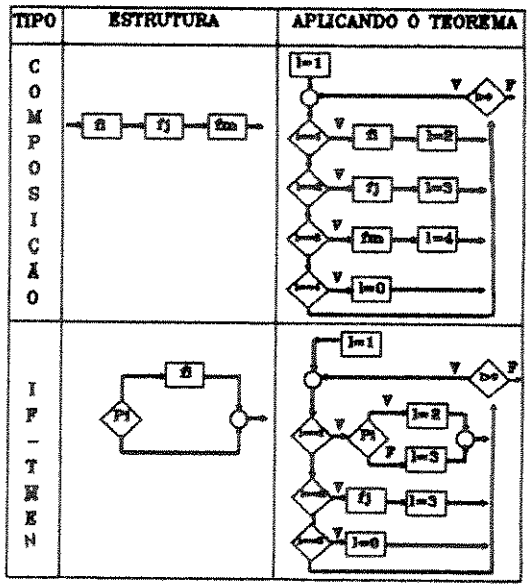

Tabela 2.1 : Teorema da Estrutura
Aplicado às Estruturas de Controle – Parte a

Estruturas de controle do tipo composição, seleção e iteração podem ser **programas próprios**, desde que não ocorram desvios para fora da estrutura. Uma chamada a um subprograma também é um **programa próprio**.

Logo, para cada estrutura de controle, e_1, que é um **programa próprio**, pode-se impor as transformações da Tabela 2.1, onde $e_1 \rightarrow e_1' = \{s_{11},\ldots,s_{1n1}\}$, $n_1 >= 1$ e s_{1j} é uma ***seqüência***, $1 <= j <= n_1$.

No entanto, estas estruturas são usadas no contexto de um programa maior, que deve ser representado em sua **Primeira Forma Normal**, cujas ***seqüências*** podem estar interligadas por desvios.

Deve-se lembrar que um desvio é um nó coletor, cuja ***seqüência*** correspondente é *"i:1=j"*, onde *i* é o número da ***seqüência*** do nó coletor e *j* o número da ***seqüência*** para onde o desvio é dirigido.

Um programa é formado por estruturas de controle, $P=\{e_1,e_2,\ldots,e_n\}$, $n >= 0$, onde a ordem de execução depende da disposição das estruturas e é ditada por uma estrutura e_j, $0 < j <= n$.

Para integrar as decomposições das estruturas em um único programa, utiliza-se o contador auxiliar *"1"*, de tal forma que ele mantenha a ordem de execução das estruturas transformadas dentro do programa. Ou seja, as ***seqüências*** s_{1j}, decorrentes da transformação de alguma estrutura e_1, devem ser numeradas pela sua ordem de ocorrência no programa, assim como as atribuições ao contador auxiliar *"1"*.

Assim, um programa $P=\{ e_1, e_2, \ldots, e_n \}$ deve ser transformado em $P' = \{ e_1', e_2', \ldots, e_n' \}$. Ou ainda, $P'=\{ s_{11}, \ldots, s_{1n1}, s_{21}, \ldots, s_{2n2}, \ldots, s_{n1}, \ldots, s_{nnn} \}$, onde s_{1j} é uma ***seqüência*** de e_1', e a cardinalidade de P' é maior ou igual à cardinalidade de P.

Generalizando as decomposições para cada estrutura de controle em um programa P qualquer, temos a **Tabela de Transformações**, Tabela 2.2.

Note-se que na **Tabela de Transformações** todo tipo de iteração é transformada em um **"do-while"**. Esta transformação tem o propósito de englobar o sub-conjunto da linguagem **C** no qual o programa estruturado será representado, que é **{ expressões e chamadas a subprogramas, "do-while", "if-else"}**. O **seqüenciamento** deve levar o programa a apresentar apenas três estruturas de controle, o que inclui a presença de um único tipo de estrutura de iteração.

Com a **Tabela de Transformações** tem-se uma referência genérica para a decomposição de qualquer **unidade de programa** em **programas primos** levando-a à sua **Primeira Forma Normal**, desde que apresentem quaisquer das estruturas de controle relacionadas, o que cobre a maior parte dos códigos fonte de produtos de software implementados em linguagens procedimentais.

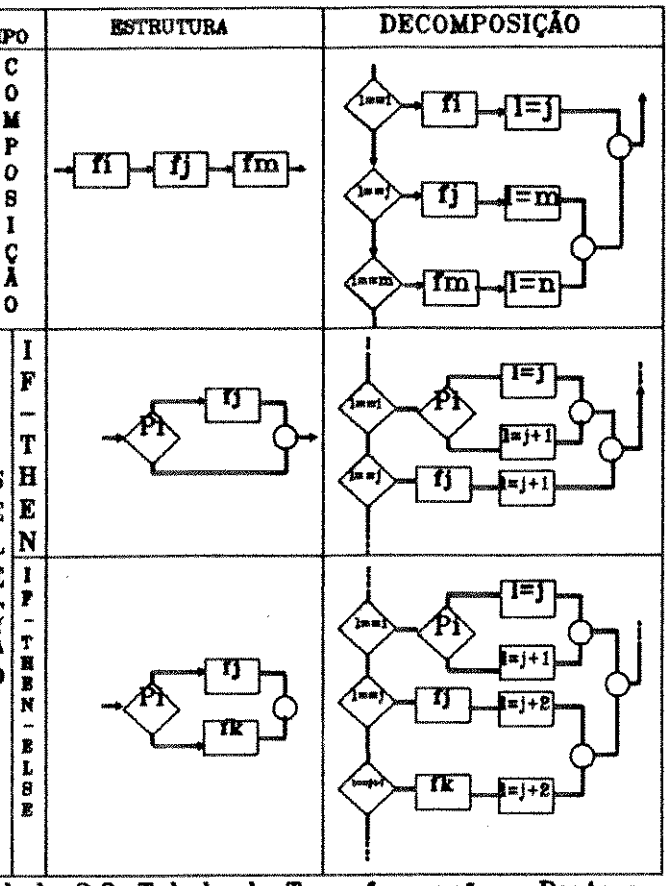

Tabela 2.2: Tabela de Transformações -Parte a

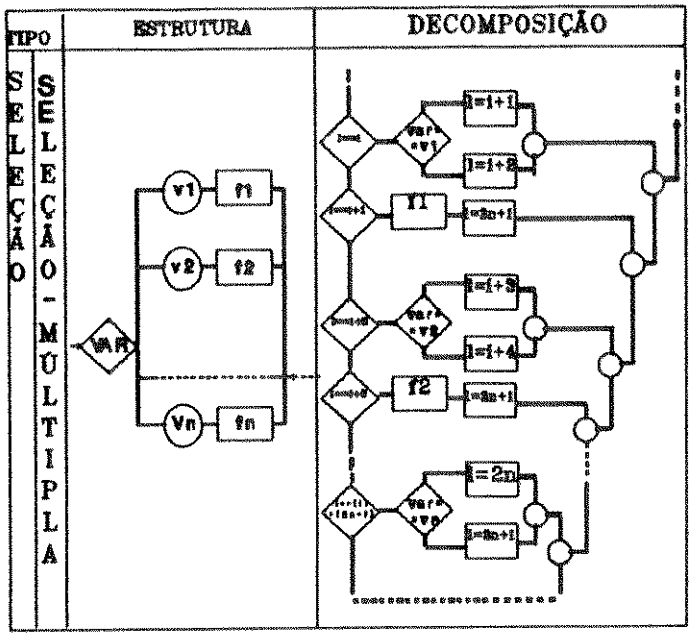

Tabela 2.2: Tabela de Transformações -Parte b

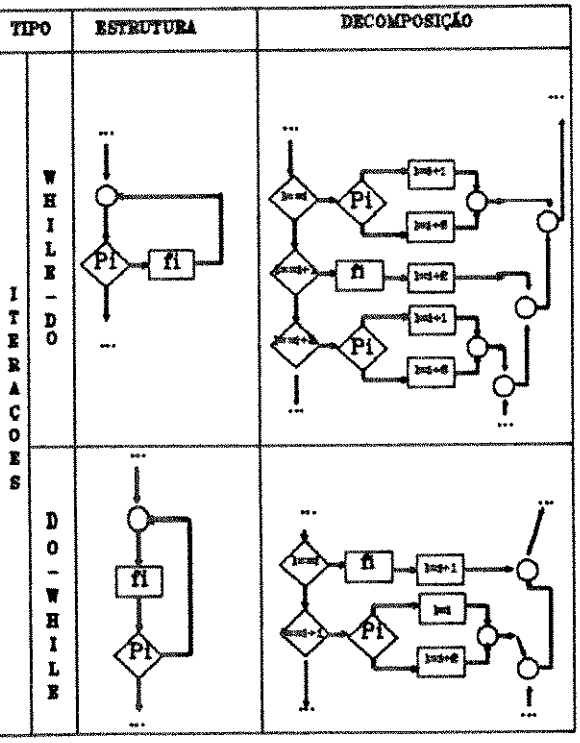

Tabela 2.2: Tabela de Transformações - Parte c

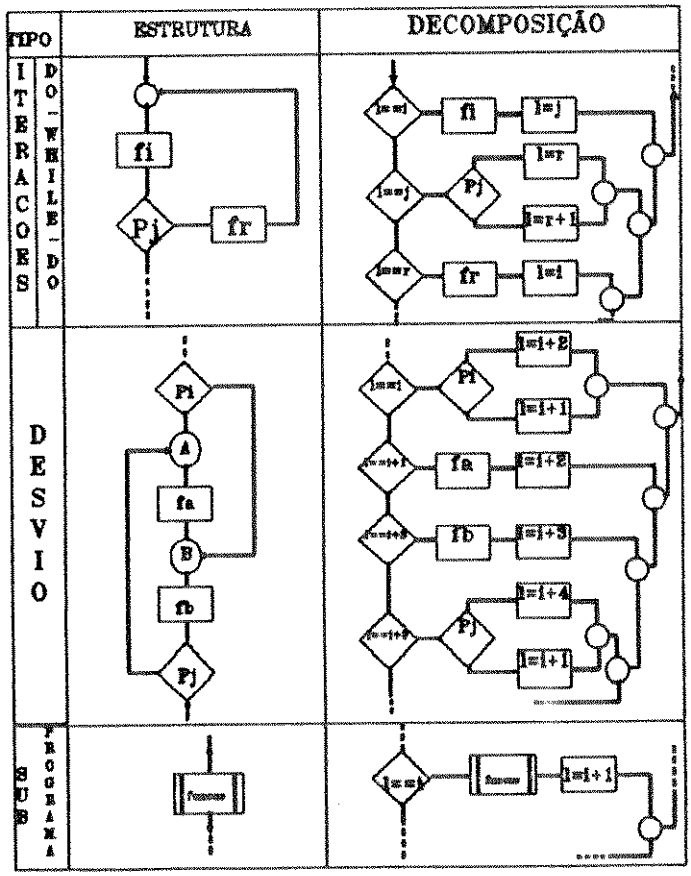

Tabela 2.2: Tabela de Transformações -Parte d

2.1.5. Estrutura de Representação - PrimeTree

O **seqüenciamento** de uma **unidade de programa**, transformação $P = \{ e_1, \ldots, e_n \} \rightarrow P' = \{ e_1', \ldots, e_n' \}$, gera uma **unidade de programa** estruturada, em sua **Primeira Forma Normal**.

Para definir operações sobre a **unidade de programa** estruturada P', tais como reconhecimento de sua hierarquia, número de ***seqüências***, tipo de **programas primos** em cada ***seqüência***, compor programas primos e outras, é necessário definir um tipo abstrato de dados que permita realizar essas operações [BARTUSSEK, 86]. Como a **unidade de programa** estruturada possui uma estrutura hierárquica [PRATT, 84], foi definido um tipo abstrato árvore [HOPCROFT, 79] denominado **PrimeTree**.

Cada nó da **PrimeTree** representa um tipo de **programa primo** contido na base fixada, {**comandos seqüenciais, if-then-else, do-while**}. O conteúdo de cada nó da árvore representa um nó funcional ou nó predicativo associado ao **programa primo**.

Como os comandos seqüenciais podem formar composições, a árvore tem um tipo de nó para esta representação; e, uma diferenciação de tipos de nós entre um comando seqüencial comum e uma chamada a um subprograma. Logo, a **PrimeTree** possui cinco tipos de nós, que são :

1. **Nó AND** : representa uma composição, ou seja um bloco de comandos executados seqüencialmente. O número de filhos de um nó do tipo **AND** é maior que zero; a única restrição é não ter um filho do tipo **AND**. O nó raiz da **PrimeTree** é do tipo **AND**.

2. **Nó OR** : representa uma seleção, ou conforme a restrição do teorema, uma seleção de duas saídas. Pode ter no máximo dois filhos. O filho da esquerda será um nó do tipo **AND**, que corresponde ao lado "**then**" de um "**if**". O filho da direita, também um nó **AND**, representará o lado "**else**" de um "**if**", e só existirá se este "**if**" possuir um "**else**" respectivo.

3. **Nó LOOP** : representa uma iteração, ou seja, um "**do-while**", e, obrigatoriamente, tem um único filho que é do tipo **AND**.

4. **Nó NILL** : um comando seqüencial simples, ou seja um nó funcional. Obrigatoriamente é filho de um nó do tipo **AND**, e não tem filhos.

5. **Nó CALL** : representa a chamada a um subprograma. Obrigatoriamente será filho de um nó do tipo **AND** e não tem filhos.

Uma ***seqüência*** na **PrimeTree** é representada por um nó do tipo OR com um filho do tipo **AND** (composição), como ilustrado na Figura 2.8. Deve-se observar que o nó do tipo **OR**, cujo conteúdo é *"I==i"*, corresponde ao número da **seqüência** na **unidade de programa**, e a composição, nó do tipo **AND**, e seus filhos contêm a funcionalidade da ***seqüência***. Nós do tipo **NILL**, com conteúdo *"I=j"*, $0 < j < n$, n ***qualquer***, correspondem às atribuições ao contador auxiliar *"I"*.

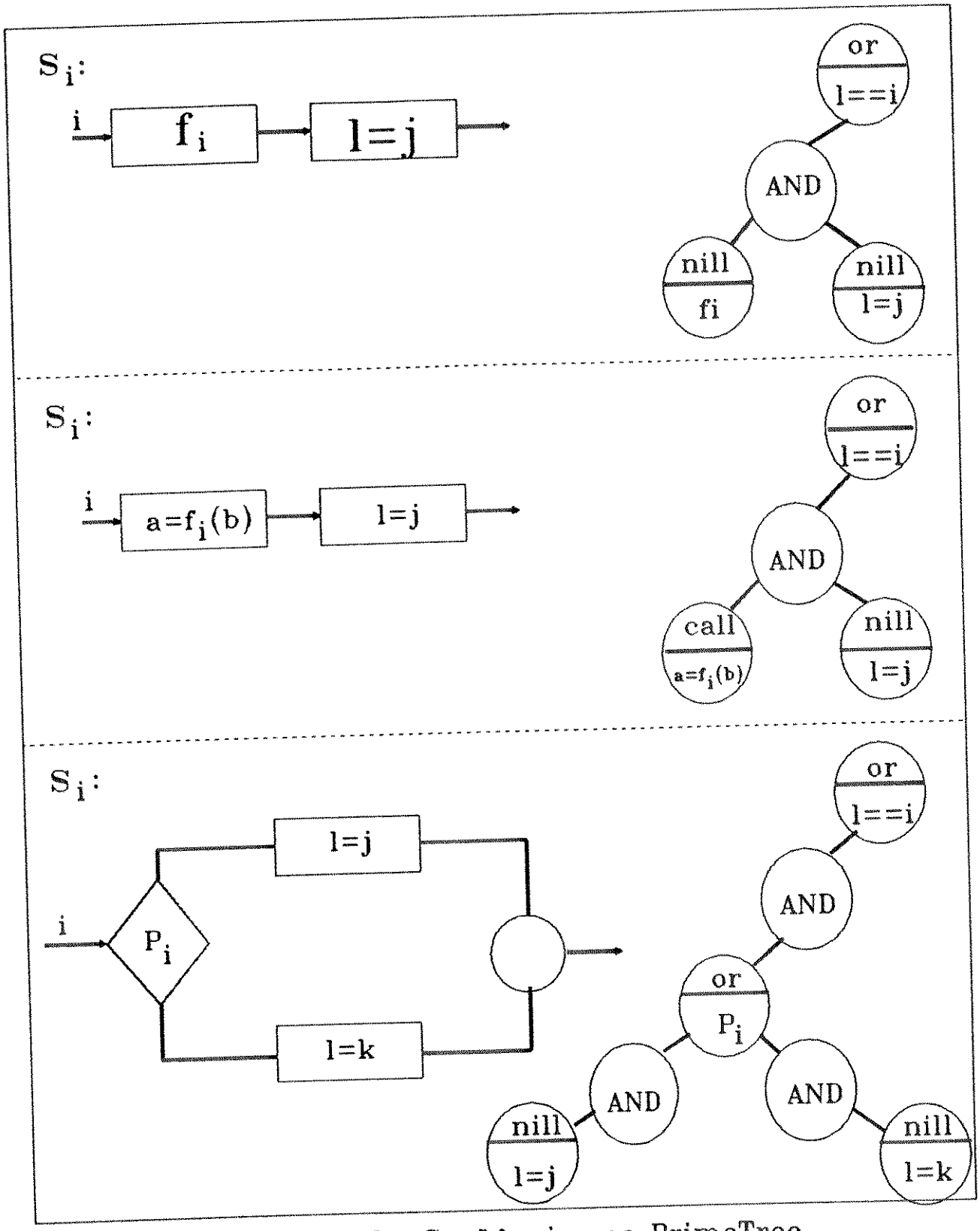

Figura 2.8 : Seqüências na PrimeTree

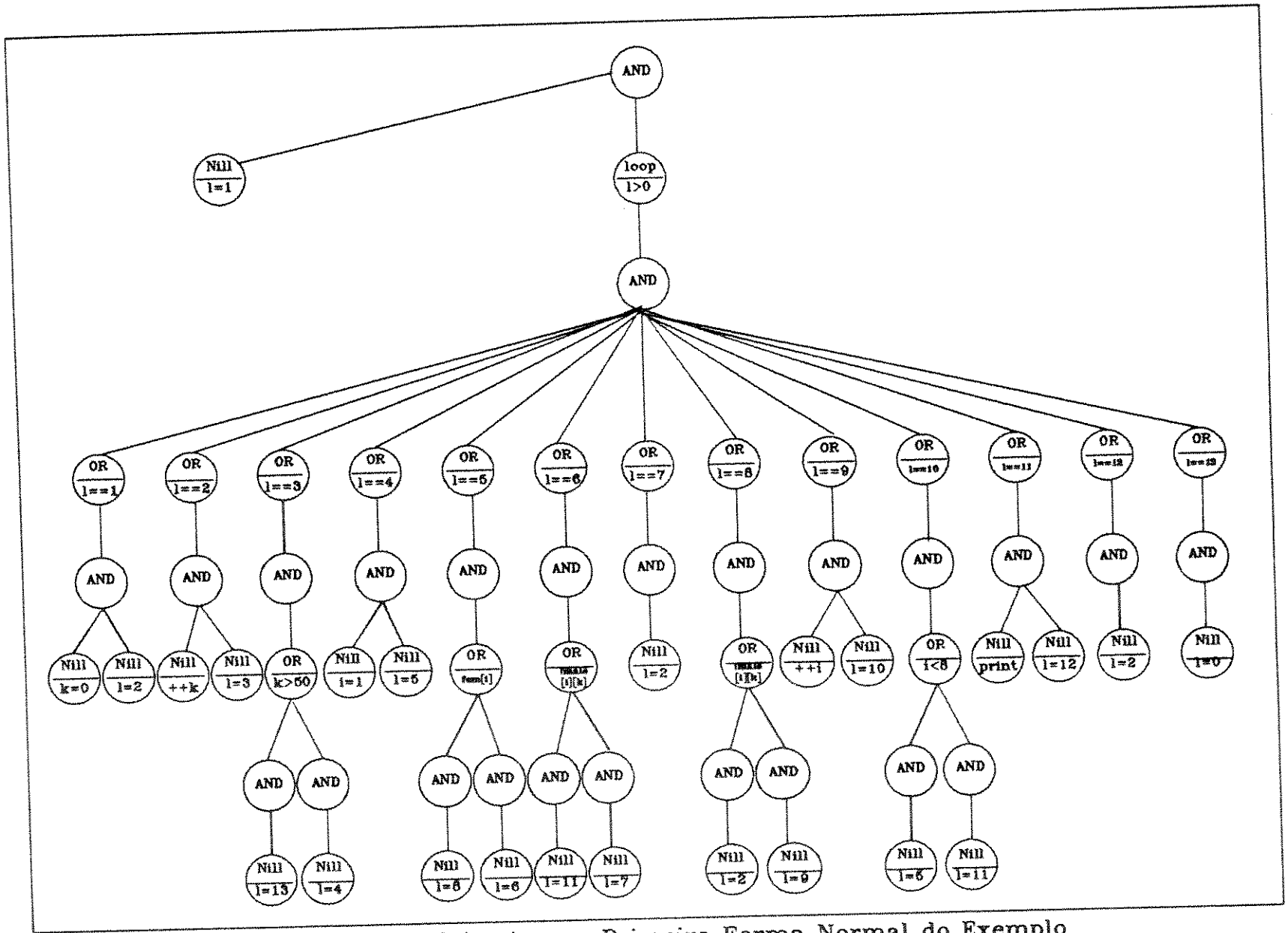

Figura 2.9 : Primetree – Primeira Forma Normal do Exemplo

A **unidade de programa** da Figura 2.6 em sua **Primeira Forma Normal** é representada pela **PrimeTree** da Figura 2.9. Toda **unidade de programa** em sua **Primeira Forma Normal** será representada por uma **PrimeTree** similar à da Figura 2.9, exceto pelo número de *seqüências* e seus conteúdos.

O **Modelo de Estruturação** é definido por algoritmos sobre o tipo abstrato **PrimeTree**, que fazem uso dos seguintes operadores :

Construtores :

PT_Cria : para criar uma **PrimeTree** com o nó raiz,que é do tipo **AND**.

PT_Destroi : para remover uma **PrimeTree** da memória.

PT_Insere : a partir de um nó já existente, cria um novo nó, como filho do primeiro.

PT_Remove : remove um único nó da **PrimeTree**.

PT_Compoem : compõe um conjunto de dois nós. O primeiro nó é do tipo **NILL** e será trocado pelos filhos do segundo nó, que é do tipo **AND**.

PT_DelRamo : elimina um ramo da árvore, dado seu nó inicial.

Observadores :

PT_Raiz : retorna o nó raiz de uma **PrimeTree**.

PT_Pai : dado um nó qualquer, retorna o nó pai do nó.

PT_Filho : dado um nó qualquer e um número inteiro "i", retorna o *i-ésimo* filho do nó.

PT_QtdFilho : retorna a quantidade de filhos de um dado nó.

PT_Localiza : localiza o nó a que corresponde a função, e/ou predicativo, de um determinado **programa primo**.

PT_NFilho : identifica o número do filho de um nó em relação ao seu nó pai.

PT_Antecessor : localiza o nó que corresponde ao **programa primo** antecessor a um dado **programa primo**.

PT_Sucessor : localiza o nó que corresponde ao **programa primo** sucessor de um dado **programa primo**.

PT_Tipo : retorna o tipo de um determinado nó.

PT_Elemento : retorna o conteúdo de um nó.

PT_UltDesc : devolve o último descendente de um ramo da **PrimeTree**.

PT_Descendente : verifica se para um conjunto de dois nós o segundo descende do primeiro.

2.2. O Modelo de Estruturação

O **Modelo de Estruturação** aqui proposto gera uma nova representação para o código fonte de um produto de software, estruturalmente padronizada e funcionalmente equivalente ao código original; tem como objetivo fornecer uma base apropriada para métodos sistemáticos de reengenharia e/ou engenharia reversa.

O próprio modelo estabelece um método sistemático, definido por algoritmos que viabilizam a implementação de ferramentas automatizadas para a estruturação. A nova representação é obtida pela decomposição de programas em **programas primos**, baseada no **Teorema da Estrutura** [LINGER, 79].

O teorema garante a transformação de uma **unidade do programa** original, isto é, um trecho de programa para o qual existe apenas um único ponto de entrada e um único ponto de saída, em uma **unidade** estruturada e funcionalmente equivalente à original. A transformação mantém como estruturas de controle apenas blocos seqüenciais, repetição e seleção simples, que formam o conjunto de estruturas de controle necessárias para garantir a construção de um programa estruturado [BOHM, 66].

Logo, para cada estrutura de controle de uma linguagem de programação foram estabelecidas transformações, com base no **Teorema da Estrutura**, às quais denominamos **seqüenciamentos**. Os seqüenciamentos, de acordo com a **Tabela de Transformações** (Tabela 2.2, Seção 2.1.4), levam cada estrutura de controle da linguagem de programação original a um conjunto de **seqüências**, que envolvem apenas três tipos de estruturas de controle.

Considerando a aplicação do modelo para códigos fonte nas linguagens de programação **COBOL**, **FORTRAN** e **C**, tem-se diferentes definições de **unidades de programa**.

Para a linguagem **C** uma **unidade de programa** é sempre uma função [KERNIGHAN, 86], para a qual sempre existe um único ponto de entrada e cujo ponto de saída não é necessariamente único nem o último comando executável da **unidade**. Logo, a **unidade de programa** pode não ser um **programa próprio**.

Para a linguagem **FORTRAN**, **unidades de programa** são o programa principal, subrotinas e funções. O mesmo problema encontrado na linguagem **C**, em relação ao ponto de saída da **unidade**, ocorre, e, além disso, as funções e subrotinas podem ter múltiplos pontos de entrada; portanto uma **unidade de programa FORTRAN** pode não ser um **programa próprio**.

Para a linguagem **COBOL**, não existe escopo de procedimentos, ou seja, todas as variáveis são de uso global e não existe uma delimitação exata para um procedimento. **Unidade de programa**, neste caso, é o programa todo e também qualquer parte do programa referenciada como subprograma (parágrafos ou seções referenciados por um comando "perform"). Trechos de programa referenciados como subprogramas podem pertencer à seqüência de execução natural do programa e, ainda, estar emaranhados por desvios, conseqüentemente podem não delimitar um programa próprio.

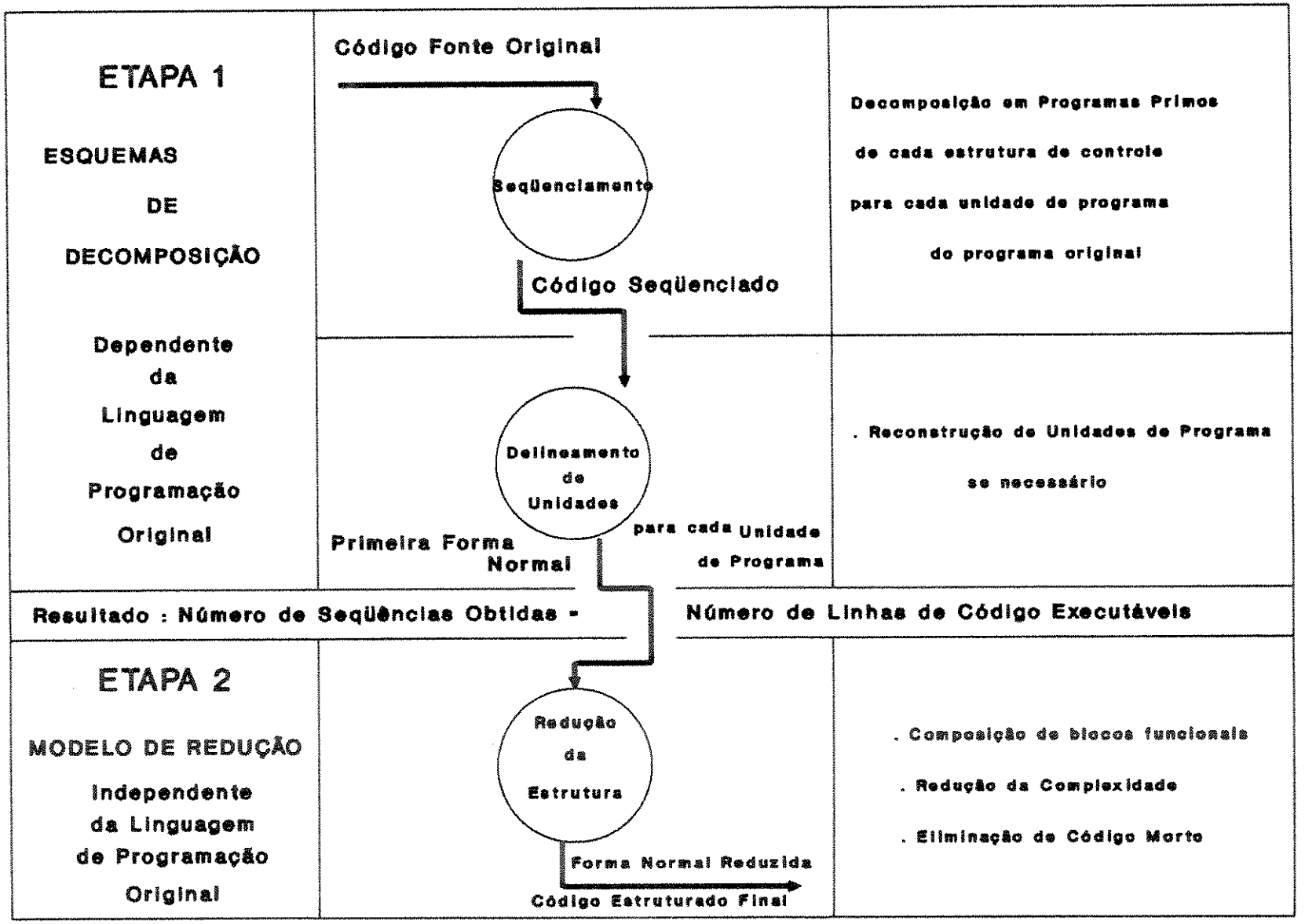

Figura 2.10 : Modelo de Estruturação Proposto

Assim, apenas o **seqüenciamento** de estruturas de controle não é suficiente para estruturar um programa, se as suas **unidades** não forem **programas próprios**. No entanto, se a linguagem de programação permite que haja entrelaçamentos entre as **unidades de programa**, o **seqüenciamento** e a disposição das **unidades** originais fornecem dados, necessários e suficientes, para verificar se isto ocorre no código fonte original. Caso existam entrelaçamentos, as **unidades** serão reconstruídas através de algoritmos; denominamos esta reconstrução de **delineamento de unidades**.

A Figura 2.10 ilustra as etapas do **Modelo de Estruturação**, onde pode ser observada a dependência da linguagem de programação original na **Etapa 1**. Esta etapa corresponde aos **esquemas de decomposição** para cada linguagem, que correspondem à decomposição do código fonte original em **programas primos** de acordo com o **Teorema da Estrutura**. Os esquemas de decomposição englobam as operações de **seqüenciamento** e os algoritmos de **delineamento de unidades** do programa.

Ao final da **Etapa 1** temos a **Primeira Forma Normal** para cada **unidade de programa** delineada a partir do código fonte original. Como as **seqüências** obtidas são as decomposições mínimas de cada estrutura de controle, o número total de **seqüências** é uma estimativa eficaz do número de linhas de código executáveis, presentes no código fonte padronizado.

Embora a **Primeira Forma Normal** do código seja um programa estruturado, ela não é cognitivamente simples, em outras palavras, não apresenta uma estrutura que possa ser funcionalmente abstraída a partir de uma simples leitura do código por um programador.

O padrão obtido é muito complexo considerando-se o número de ciclos do programa [McCABE, 76] e as **seqüências** obtidas são fortemente acopladas. O acoplamento é dado pelo uso de uma única variável auxiliar para seqüenciar o código, o que dificulta o particionamento do mesmo quando realizada uma análise de interdependência entre variáveis ([GATTAZ, 84] e [REZENDE, 92]).

A complementação do **Modelo de Estruturação**, que na Figura 2.10 corresponde à **Etapa 2**, é um processo de simplificação da estrutura obtida e minimização do uso da variável auxiliar. A base do **Modelo de Redução** é o processo de simplificação da **Primeira Forma Normal**, a composição de **programas primos** proposta por Linger e Mills (Seção 2.1.3); que reduz a complexidade obtida na **Primeira Forma Normal** em relação ao número de ciclos. No entanto, as composições são realizadas sem repetição de **seqüências** do programa, ou seja, sem replicação de código; além disso, são identificados blocos de iteração. As composições também permitem identificar e eliminar **código morto**, partes do programa para as quais não há um caminho no fluxo de execução do programa [MILLER, 91].

A redução da estrutura de controle, **Etapa 2**, fornece a representação estruturada final do código original, que pode ser representada por uma linguagem de programação estruturada qualquer, desde que suporte as três estruturas de controle impostas pelo **Teorema da Estrutura**. Todavia, neste trabalho, a representação do código estruturado final será feita através do sub-conjunto da linguagem **C**, definido anteriormente.

A representação final, obtida para cada **unidade de programa**,

não apresentará a variável auxiliar empregada no **seqüenciamento** se e somente se o código original possuir **unidades** bem delineadas e apenas estruturas de controle do mesmo tipo que as impostas. Caso contrário, a presença da variável auxiliar é a única forma de garantir uma representação estruturada e funcionalmente equivalente ao código original.

No Capítulo 3 são descritos os **esquemas de decomposição** para as linguagens **COBOL**, **FORTRAN** e **C**. O **Modelo de Redução**, **Etapa 2** do **Modelo de Estruturação**, que depende exclusivamente do resultado obtido na **Etapa 1**, é detalhado no Capítulo 4.

3. ESQUEMAS DE DECOMPOSIÇAO — LINGUAGENS PROCEDIMENTAIS

Um esquema de decomposição é a transformação de um programa *P*, implementado em uma linguagem de programação procedimental *L*, para sua **Primeira Forma Normal**, *P'*. A transformação envolve os **seqüenciamentos** para cada estrutura de controle da linguagem *L* e, se a linguagem permite emaranhar procedimentos, o **delineamento de unidades de programa**.

Para aplicar o **Teorema da Estrutura** a um programa *P* qualquer é necessário que *P* seja um **programa próprio**, ou seja, corresponda a uma **unidade completa de programa**. Se *P* é um **programa próprio**, os **seqüenciamentos** das estruturas de controle, de acordo com a Tabela de **Transformações** (Tabela 2.2), são necessários e suficientes para obter *P'*.

O exemplo a seguir mostra um **programa próprio**, implementado em uma pseudo-linguagem qualquer :

```
passo1.
     leia(a,b,c)
     delta = b*b + 4*a*c
     se ( delta < 0 ) {
          vá para erro
     }
     delta = raizquadrada(delta)
     se ( delta = 0 ) {
          raiz1 = raiz2 = -b / ( 2 * a )
     }
     cc {
          raiz1 = ( -b + delta ) / ( 2 * a )
          raiz2 = ( -b - delta ) / ( 2 * a )
     }
     imprima(raiz1,raiz2)
     vá para fim
erro.
     imprime ( mensagem de erro )
fim.
     saída()
```

A **PrimeTree** resultante do **seqüenciamento** do programa acima é a **Primeira Forma Normal**; a **PrimeTree** é ilustrada pela Figura 3.1 e o seqüenciamento do programa é o seguinte :

Seqüenciamento:

```
1  : leia(a,b,c) ;                          1 =  2
2  : delta = b*b - 4*a*c ;                  1 =  3
3  : se (delta < 0)                         1 =  4
     cc                                     1 =  5
4  :                                        1 = 13
5  : delta = raizquadrada(delta) ;          1 =  6
6  : se (delta == 0 )                       1 =  7
     cc                                     1 =  9
7  : raiz1=raiz2= -b / (2*a) ;              1 =  8
8  :                                        1 = 11
9  : raiz1 = (-b+delta) / (2*a) ;           1 = 10
10 : raiz2 = (-b-delta) / (2*a) ;           1 = 11
11 : imprima(raiz1,raiz2) ;                 1 = 12
12 :                                        1 = 14
13 : imprime (mensagem de erro) ;           1 = 14
14 : saida() ;                              1 =  0
```

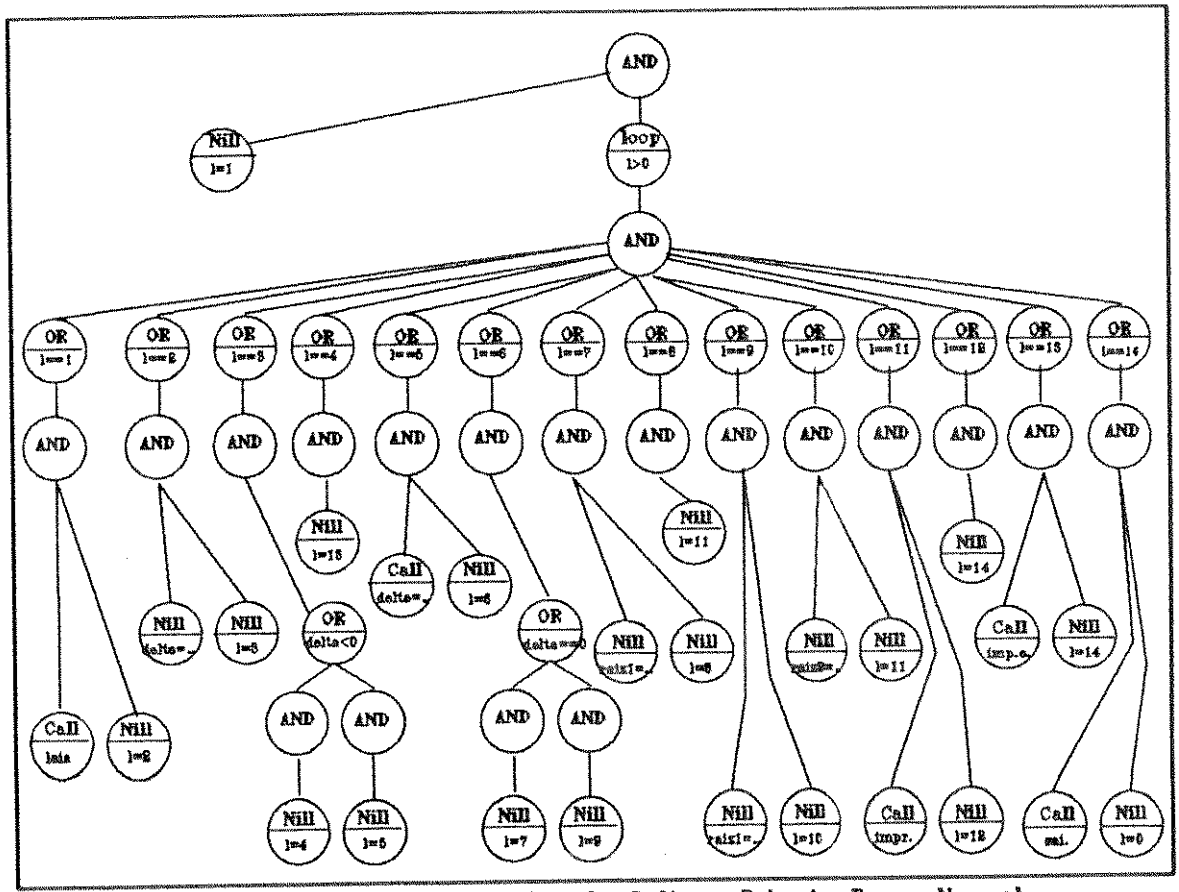

Figura 3.1 : PrimeTree do Pseudo-Código - Primeira Forma Normal

Mesmo que uma **unidade de programa** possua mais de um ponto de saída, ou seja, mais de um comando de retorno ao seu ponto de chamada, o simples **seqüenciamento** pode transformá-la em um **programa próprio**.

O exemplo a seguir mostra um caso de programa com vários pontos de retorno e seu **seqüenciamento** :

Programa :

```
int compara (x,y)
int x;
int y;
{
     if ( x > y )
          return(-1);
     else {
          if ( x > y )
               return(1);
          else
               return(0);
     }
}
```

Seqüenciamento :

```
1  : if ( x < y )   l = 2;
     else           l = 3;
2  : fnc_return= -1;l = 0;
3  : if ( x > y )   l = 4;
     else           l = 5;
4  : fnc_return = 1;l = 0;
5  : fnc_return = 0;l = 0;
```

O **seqüenciamento** para um comando de retorno, saída de um subprograma, é definido aqui como uma única **seqüência** s_1, $l > 0$, do tipo **desvio**, para a qual a **seqüência** sucessora será 0; ou seja, s_1 é composta por um nó do tipo **NILL** com conteúdo *"l=0"*, que é um desvio para o fim do programa. O comando de retorno será inserido como um único nó irmão à direita do nó do tipo **LOOP** (inserido pela normalização). A **PrimeTree**, resultante do **seqüenciamento** do exemplo anterior, é ilustrada na Figura 3.2.

O caso do exemplo anterior é muito comum em programas implementados na linguagem de programação C, cujo esquema de decomposição é apresentado na Seção 3.3.

No entanto, se a linguagem de programação permite que haja mais de um ponto de entrada em um programa, como é o caso de **FORTRAN** (Seção 3.2), a transformação do programa para sua **Primeira Forma Normal** é mais complexa.

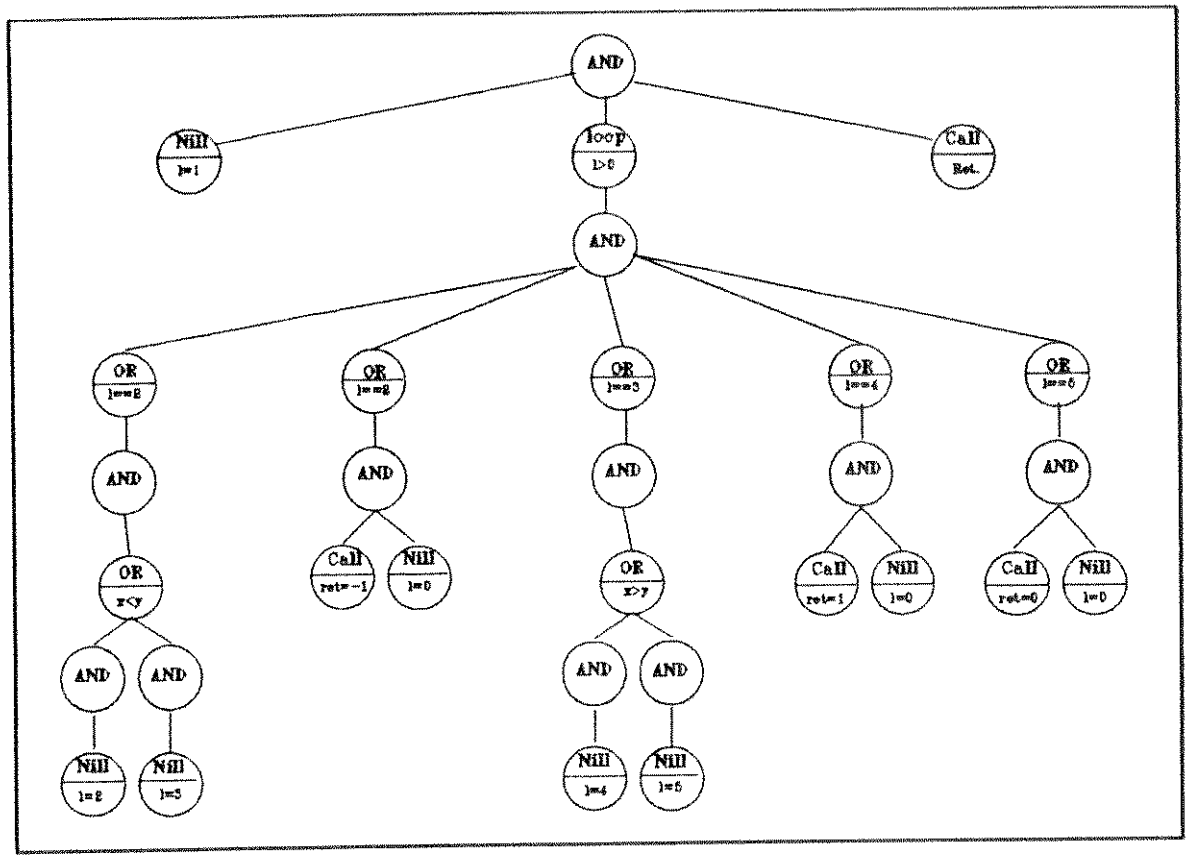

Figura 3.2 : PrimeTree - Sequenciamento do Programa C com 3 "return's"

Um programa que possui mais de um ponto de entrada define tantos programas quantos forem os pontos de entrada. Se "n" é o número de pontos de entrada de um programa então existem "n" programas sobrepostos que compartilham parte de suas funcionalidades, e, ainda, essas sobreposições podem estar emaranhadas através de desvios.

Logo, se ocorrem sobreposições funcionais é necessário reconstruir cada subprograma, de modo a transformá-los em **unidades de programa**, que sejam **programas próprios**. O **delineamento de unidades de programa** para a linguagem **FORTRAN**, Seção 3.2, mostra como esse tratamento é realizado.

Programas em linguagens cujo escopo de procedimentos não é bem delimitado, podendo ocorrer desvios entre eles, sobreposições funcionais ou mesmo redefinições funcionais, exigem uma análise e tratamento mais rigorosos. A linguagem de programação **COBOL** permite escrever programas com todo este emaranhado de problemas.

Normalizar um programa implementado em **COBOL** consiste em **seqüenciar** toda sua estrutura de controle e analisar todas as possíveis intersecções (sobreposições funcionais) entre suas **unidades**, para que se possa reconstruí-las como **programas próprios**; o **esquema de decomposição** para **COBOL** é apresentado na Seção 3.1.

Nas próximas seções são apresentados **esquemas de decomposição** para as linguagens **COBOL**, **FORTRAN** e **C**. Nessas seções não há a preocupação com as variações existentes entre as diversas versões e compiladores de cada linguagem, e sim com as principais estruturas de controle encontradas nas versões mais utilizadas.

O enfoque principal é mostrar os **esquemas de decomposição** e os problemas relacionados à automatização do processo de estruturação, devido às características de cada linguagem.

3.1. Esquema de Decomposição para a Linguagem COBOL

Programas implementados na linguagem de programação **COBOL** geralmente são muito mal estruturados, pois a linguagem não possui uma caracterização de escopo de procedimentos. A divisão de programas é feita através do uso de parágrafos e/ou seções [PRATT, 84], e a presença de desvios entre essas divisões também é uma construção válida para a linguagem.

As estruturas de controle incluem o comando "**goto**", uma construção de "**if-then-else**" restrita, e o comando "**perform**" que pode ser usado tanto para uma chamada simples a um subprograma quanto como um comando iterativo. O tratamento de expressões é restrito e não há recursos para definições de novas funções pelo programador [PRATT, 84]; existe, porém, um tratamento de exceções que pode piorar a estrutura provocando novos desvios do fluxo de controle (vide **Apêndice A**).

Uma **unidade de programa**, para programas em linguagem COBOL, pode ser considerada o programa todo e/ou parágrafos e seções referenciados por um comando "**perform**". Porém, a linguagem também permite que ocorram desvios para fora de blocos de comandos reunidos em parágrafos e/ou seções e desvios que variam em tempo de execução (comando **ALTER**).

Se um programa COBOL for considerado uma única **unidade de programa**, o **seqüenciamento** de sua estrutura de controle pode não levar diretamente à sua **Primeira Forma Normal**, pois podem existir várias **unidades** no mesmo programa além de entrelaçamentos e/ou sobreposições entre elas.

Para melhor ilustrar o problema consideremos o exemplo a seguir, onde ocorre a definição de algumas **unidades de programa** e entrelaçamentos entre elas :

```
procedure division.
programa section.
par1.
        move 0 to p.
        move 1 to q.
        perform b.
        perform c.
        perform d.
        perform e.
        perform tudo.
        stop run.
tudo section.
a.
        move 0 to q.
        move 0 to p.
b.
        if p = 1
            go a.
        if q = 1.
            go d.
c.
        add 1 to i.
d.
        if q = 1
            go a.
e.
        display "i = " i.
```

Em princípio, todo comando *"perform"* deveria executar um bloco (parágrafo ou seção) de comandos inteiro e voltar ao comando seguinte a ele, sem que houvesse desvios. A funcionalidade simularia uma chamada a um subprograma, tendo como ponto de retorno o último comando do bloco executado.

No entanto, no exemplo anterior ocorrem desvios misturados a *"perform´s"*, que não invalidam a execução dos parágrafos, pois passam pelo seu ponto de retorno [MICROSOFT, 80]. O uso de desvios para fora de parágrafos, que são também executados através do comando *"perform"*, pode não causar erros de execução do programa; desde que esses desvios permitam que o fluxo de execução volte ao ponto de retorno do *"perform"*, ou encerrem a execução do programa.

Se o código do exemplo for diretamente **seqüenciado** e dividido em **PrimeTrees**, a partir de cada parágrafo ou da seção referenciados por um comando **"perform"**, as **PrimeTrees** obtidas não seriam **programas próprios**, pois não existiria um caminho do ponto de entrada ao ponto de saída para qualquer nó da **PrimeTree**.

A Figura 3.3 ilustra o grafo de fluxo de controle do programa do exemplo (obtidos com o módulo "chanomat" da ferramenta "POKE-TOOL" [LEITAO, 92]), onde os arcos dirigidos indicam o fluxo de controle e cada nó rotulado um conjunto de comandos seqüenciais.

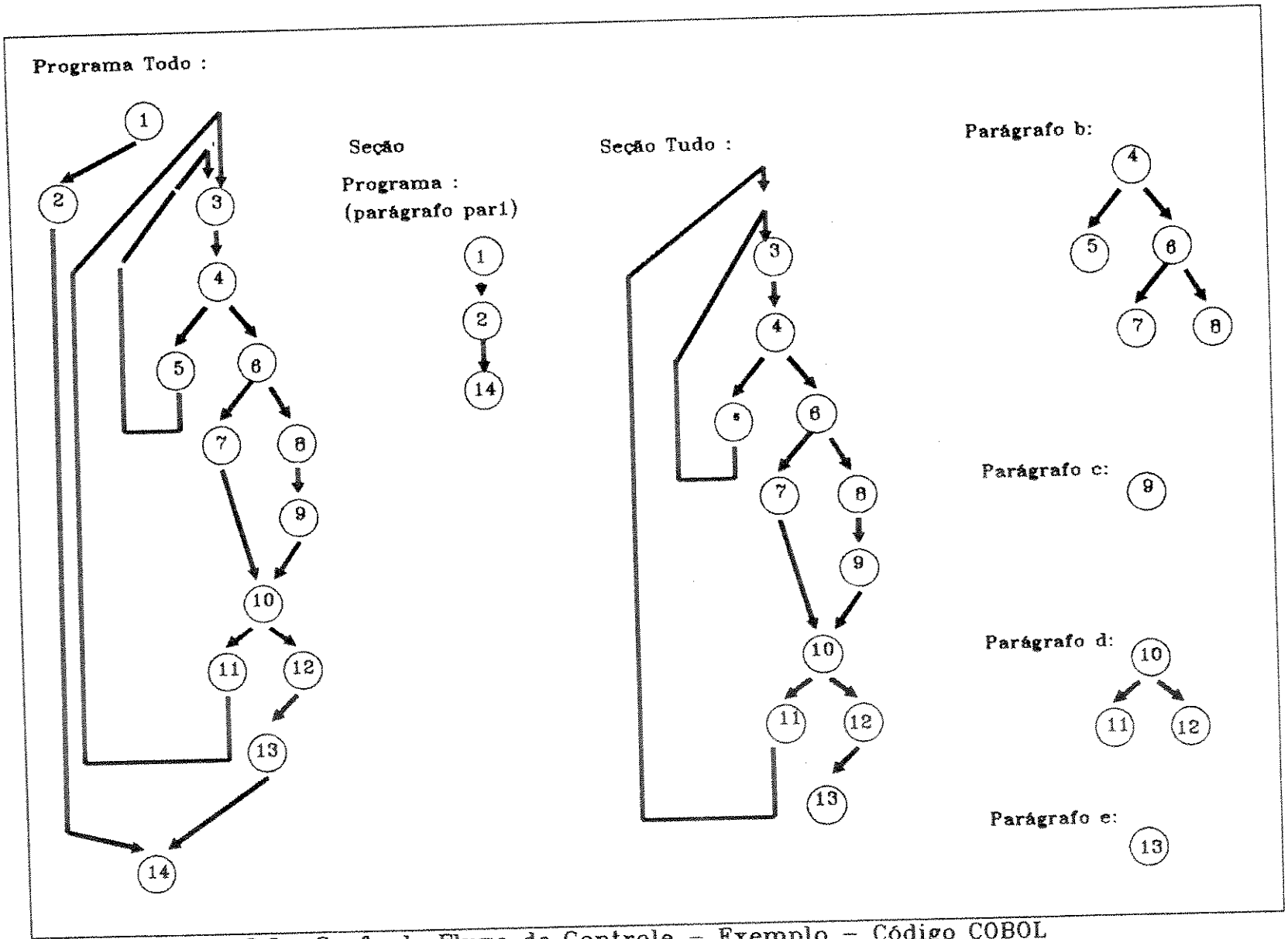

Figura 3.3 : Grafo de Fluxo de Controle – Exemplo – Código COBOL

Nota-se no grafo da Figura 3.3 que o parágrafo "**par1**" é uma **unidade de programa**, é bem delimitado, tem um único ponto de entrada, que é o nome do parágrafo, e um único ponto de saída, que é o comando "**stop run**". A seção "**tudo**" também é uma **unidade**, dado que os desvios nela presentes são todos internos ao seu escopo. O mesmo pode ser observado para os parágrafos "**a**", "**c**" e "**e**". No entanto, os parágrafos "**b**" e "**d**" possuem desvios para fora de seus limites, além de serem referenciados por um comando "**perform**". Logo, os parágrafos "**b**" e "**d**" deverão ser compostos com as partes do programa para as quais eles desviam.

A Figura 3.4 mostra como o programa poderia ser composto a partir de uma **unidade de programa** única e uma conseqüente subdivisão e recomposição de suas partes. Cada retângulo rotulado corresponde a um trecho de código de um parágrafo e/ou seção, cujos nomes originais correspondem aos rótulos dos retângulos e os arcos dirigidos são os desvios que aparecem entre os trechos de código.

Na Fase 1 da Figura 3.4 o programa todo é seqüenciado como uma única **unidade de programa**, ou ainda, uma única **PrimeTree**. Na segunda fase o programa seqüenciado é dividido de acordo com as referências aos parágrafos e/ou seções através do uso de comandos "**perform**" no código original, o que corresponde ao primeiro **delineamento de unidades de programa** e à criação de novas **PrimeTrees**. As novas **PrimeTrees** são obtidas a partir de cópias da **PrimeTree** "**main**" das **seqüências** que as compõem. Cada subdivisão dos retângulos, como "**b1**", "**b2**" e "**b3**" tem o intuito de ilustrar as partes do código que estão entre os pontos de desvio dentro de um mesmo trecho da unidade - parágrafo e/ou seção.

Na Fase 3 da Figura 3.4 as **unidades** que possuem desvios são recompostas obedecendo-se o **seqüenciamento** da primeira fase; ou seja, as **PrimeTrees** da segunda fase são ampliadas possuindo novos nós que são cópias dos originais. O resultado da terceira fase é a divisão do *código seqüenciado em* ***unidades*** *de fato*, isto é, **programas próprios**, pois todas as **seqüências** que possam ser executadas ao ativar a **unidade** estão contidas na **unidade**.

Assim, ao final da terceira fase, tem-se sete **PrimeTrees**, que correspondem às **unidades de programa** definidas no programa original, reconstruídas como **programas próprios** e representadas na **Primeira Forma Normal**; são elas :

. "**main**" : corresponde ao **seqüenciamento** de todo o programa, como uma **unidade** única,

. "**tudo** " : corresponde ao **seqüenciamento** da seção "**tudo**" como uma **unidade** única,

. "**a**" : corresponde ao **seqüenciamento** do parágrafo "**a**", que é referenciado por um comando "**perform**",

. "**b**" : corresponde ao **seqüenciamento** do parágrafo "**b**", referenciado por "**performs**" e reconstruído a partir de todas as **seqüências** que definem seus caminhos de execução,

. "**c**" : corresponde ao **seqüenciamento** do parágrafo "**c**", referenciado por um comando "**perform**",

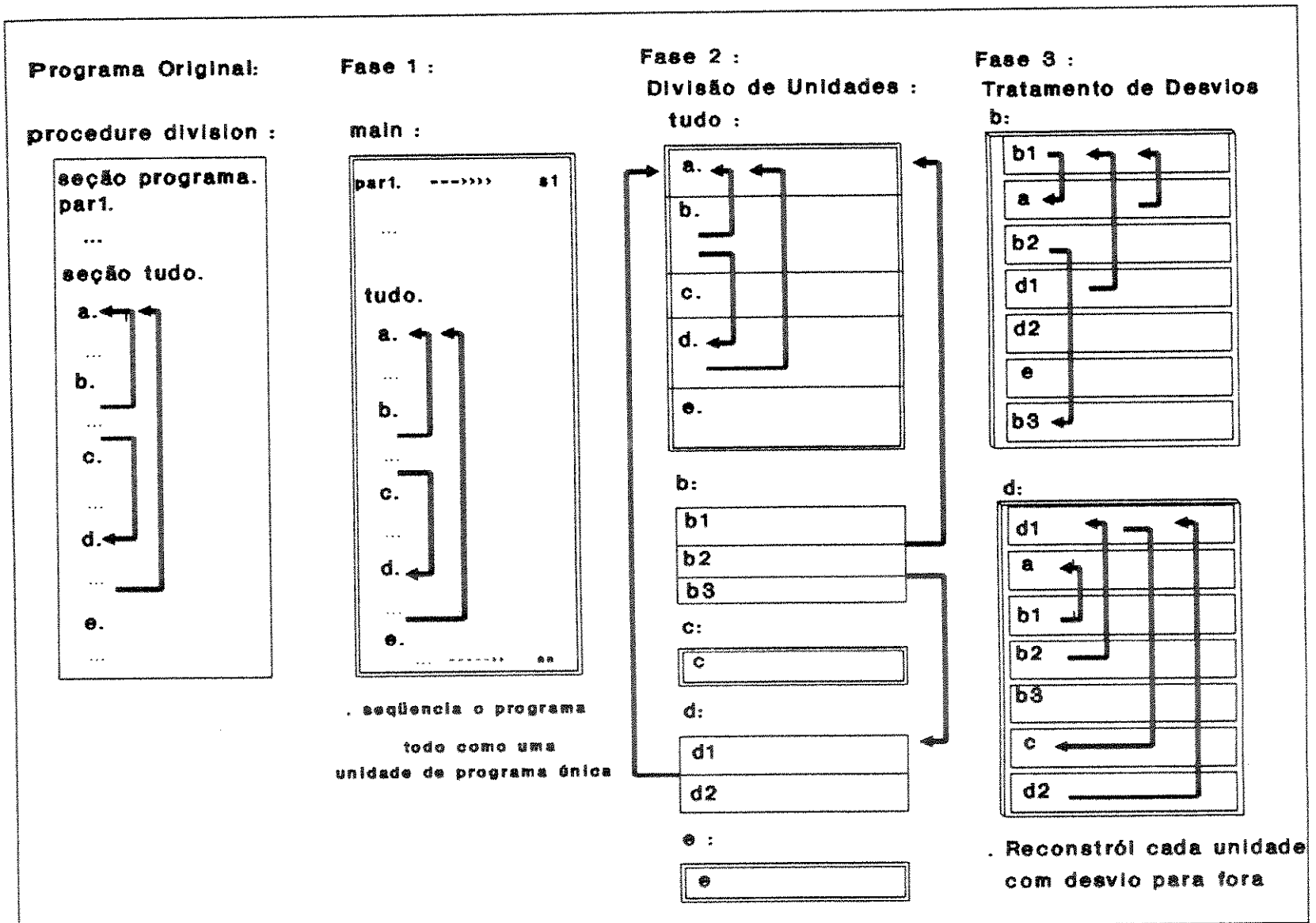

Figura 3.4 : Esquema de Decomposição Generalizado – COBOL

. "d" : corresponde ao **seqüenciamento** do parágrafo "d", referenciado por "performs" e reconstruído a partir de todas as **seqüências** que definem seus caminhos de execução, e

. "e" : corresponde ao **seqüenciamento** do parágrafo "e", referenciado por um comando "perform".

Assim, o esquema geral de decomposição para um código fonte linguagem COBOL precisa prever transformações sobre as PrimeTre obtidas pelo **seqüenciamento**, de modo que elas possam tornar- **programas próprios**; para isto o esquema foi dividido nas três fas detalhadas a seguir.

Fase 1 : Seqüenciamento do Programa Original e Tabulação de Seus Rótulos

Dado um programa *P*, com seu código fonte na linguagem COBC transformá-lo em $P' = \{s_1, s_2, \ldots, s_n\}$, onde :

P' : uma única PrimeTree, composta por todas **seqüências** de *P*

s_i : **seqüência** resultante do **seqüenciamento** de algu linha de código de *P*, onde $i=1,..,n$. Os **seqüenciament** são obtidos de acordo com a proposta do **Apêndice A**.

O **seqüenciamento** do programa original, transformação *P* -> *I* independe da existência de divisões em *P*; ou seja, não considera existência de parágrafos e/ou seções no programa original.

Para que possam ser identificadas novas PrimeTrees, a partir *P'*, paralelamente à sua construção, é montada uma tabela de rótulos programa, denominada **Ro**, e definida do seguinte modo :

Ro : Tabela de Rótulos

rótulo	início	fim	unidade
r1	s11	s1n1	u1
r2	s21	s2n2	u2
...	...	...	...
rn	sn1	snnn	un

Onde,

r_i : nome do *i-ésimo* rótulo, que pode ser um parágra ou uma seção

s_{11} : número da **seqüência** inicial do rótulo

s_{ini} : número da **seqüência** final do rótulo

u_i : indica se para o rótulo r_i existe um comando em *P* do tipo "**perform** r_i", isto é, se $u_i = 1$, então existe um tal comando em *P*, logo, **ri** deverá se tornar uma **unidade de programa.**

Para complementar o **seqüenciamento** e a montagem da tabela **Ro**, deve-se assumir que :

1. O bloco principal de execução, isto é, a **PrimeTree** P', denominar-se-á "**PrimeTree main**", e não sofrerá alterações durante o restante do processo de obtenção da **Primeira Forma Normal.**

2. Quando no código original, programa *P*, existir uma seção ("**section**"), a linha correspondente à sua declaração será transformada em uma **seqüência** do tipo *"if (1 == i) 1 = i + 1"*, ou seja, o primeiro parágrafo da seção corresponderá à **seqüência** " *i + 1* " e a seção à **seqüência** " *i* ".

3. A cada final de parágrafo e/ou seção deverá ser incluída uma **seqüência** a mais, que terá conteúdo *"if (1 == s_{ini}) 1 = s_{ini+1}"*, e será associada ao ponto de retorno da **unidade**. Por exemplo :

```
par1.
     add 1 to i.
     move 0 to k.
par2.

Seqüenciamento :

1 : i++; 1 = 2;
2 : k = 0; 1 = 3;
3 : 1 = 4; (última seqüência de par1)
4 :        (primeira seqüência de par2)
```

4. Se o comando *"perform"*, que originar um rótulo na tabela **Ro**, contiver uma cláusula *"through"*, então o nome do rótulo na tabela será uma justaposição dos nomes dos parágrafos e/ou seções envolvidas na cláusula. Além disso o número da **seqüência** inicial será o mesmo que o da primeira **seqüência** do primeiro parágrafo e o número da **seqüência** final o da última **seqüência** do último parágrafo. Por exemplo, no caso do comando *"perform c through d"*, então o nome do rótulo será *"c_d"* e o número da **seqüência** inicial será a primeira **seqüência** de *"c"* e o da final o da última **seqüência** de *"d"*.

O **seqüenciamento**[1] para o programa do exemplo (dado no início da seção) que origina a **PrimeTree** P' (**PrimeTree** "**main**") e a Tabela de Rótulos **Ro** são os seguintes :

---

1 Os seqüenciamentos são obtidos de acordo com as sugestões do Apêndice A.

Seqüenciamento :

```
(programa section)  1  :                          1=2;
(par1)              2  : p=0;                     1=3;
                    3  : q=1;                     1=4;
                    4  : b();                     1=5;
                    5  : c();                     1=6;
                    6  : d();                     1=7;
                    7  : e();                     1=8;
                    8  : tudo();                  1=9;
                    9  :                          1=10;
                    10 :                          1=0;
(tudo section)      11 :                          1=12;
(a)                 12 : q=0;                     1=13;
                    13 : p=0;                     1=14;
(fim de a)          14 :                          1=15;
(b)                 15 : if (p==1)                1=16
                         else                     1=17;
(goto a)            16 :                          1=12;
                    17 : if (q==1)                1=18
                         else                     1=19;
(goto d)            18 :                          1=22;
(fim de b)          19 :                          1=20
(c)                 20 : ++i;                     1=21;
(fim de c)          21 :                          1=22;
(d)                 22 : if (q==1)                1=23
                         else                     1=24;
(goto a)            23 :                          1=12;
(fim de d)          24 :                          1=25;
(e)                 25 : display("i=",i);         1=26;
(fim de e)          26 :                          1=0;
```

Tabela 3.1 : Tabela de Rótulos Ro

rótulo	início	fim	unidade
programa	1	10	0
par1	2	10	0
tudo	11	26	1
a	12	14	0
b	15	19	1
c	20	21	1
d	22	24	1
e	25	26	1

Se na Tabela de Rótulos **Ro** não existisse um único elemento, de i = 1 até **n**, para o qual $u_i = 1$, então o processo de normalização do código poderia ser encerrado; isto é, a **PrimeTree** *P'* seria a **Primeira Forma Normal de** *P*.

Deve-se notar que a **PrimeTree "main" é um programa próprio**, todas as seqüências que fazem parte do programa estão contidas em seu escopo e ela corresponde à **Primeira Forma Normal** da principal **unidade do programa**.

Fase 2 : Divisão das Unidades e Tabulação de Desvios

A partir da **PrimeTree** *P´* serão geradas novas **PrimeTrees** para cada rótulo ao qual corresponda, no programa original, um comando do tipo *"perform* r_1*"*, ou ainda, para $i=1,..,n$, se $u_1 = 1$ então será gerada uma **PrimeTree** $P_1' = \{s_{11},...,s_{1n1}\}$.

A **seqüência** s_{1n1} de uma **PrimeTree** P_1', na **PrimeTree** "**main**" corresponde ao ponto de retorno do parágrafo e/ou seção e é composta por um nó do tipo **NILL** com conteúdo "**l=**s_{1n1}**+1**". No entanto, na nova **PrimeTree** P_1' este conteúdo será trocado por "**l=0**"; ou seja, esta será a última **seqüência** executada em P_1', de acordo com o processo da **Fase 1**.

Nesta segunda fase, são identificados os desvios para fora das unidades. Entenderemos como **desvio** a existência de um nó do tipo **NILL**, em uma **seqüência** s_{1j}, $1 <= j <= n_1$, com conteúdo "**l=k**", tal que :

. $0 < k < s_{11}$, então existe um desvio para uma **seqüência** anterior à primeira **seqüência** de P_1'

. $(k > s_{1n1})$, então existe um desvio para uma **seqüência** sucessora à última **seqüência** de P_1'.

Paralelamente à geração das novas **PrimeTrees** P_1', será construída uma tabela de desvios D_1, definida do seguinte modo :

D_1 : Tabela de Desvios

PrimeTree	início	fim	desvio-frente	desvio-trás
P_1'	s_{11}	s_{1n1}	$gf_{11},...,gf_{1f1}$	$gt_{11},...,gt_{1t1}$
P_2'	s_{21}	s_{2n2}	$gf_{21},...,gf_{2f2}$	$gt_{21},...,gt_{2t2}$
...	...	...	...	...
P_n'	s_{n1}	s_{nnn}	$gf_{n1},...,gf_{nfn}$	$gt_{n1},...,gt_{ntn}$

Onde,

P_1' : nome da *i-ésima* **PrimeTree** gerada

s_{11} : número da **seqüência** inicial da **PrimeTree** P_1'

s_{1n1} : número da **seqüência** final da **PrimeTree** P_1'

gf_1 : conjunto de desvios encontrados na **PrimeTree** P_1', tais que $gf_{1j} > s_{1n1}$, para $j = 1, .., f_1$; onde f_1 é o tamanho do conjunto de desvios para a frente..

gt_1 : conjunto de desvios encontrados na **PrimeTree** P_1', tais que $gt_{1j} < s_{1n1}$, para $j = 1, .., t_1$; onde t_1 é o tamanho do conjunto de desvios para trás.

Vale observar que *P´*, a **PrimeTree** resultante da **Fase 1**, não contém desvios do tipo de gf_{1j} e gt_{1j}; pois contém todas as **seqüências** do programa original, por este motivo é mantida intacta e considerada a **PrimeTree** principal no **esquema de decomposição**.

O algoritmo[2], a seguir, é usado para construir as novas **PrimeTrees** $P_i´$, $i = 2,...,n$, e a tabela D_1 :

```
. para i = 2,..,n ,onde n é o número de rótulos em Ro :
    . se u1 = 1 :
        . Criar a PrimeTree P1´ => PT_Cria(r1)
        . Copiar para a tabela D1 as
              informações
              do nome do rótulo r1 como o nome
              de
              P1´, s11 e sin1 como o número das
              seqüências inicial e final de P1´
        . Para j = 1,..., n1; onde s1j é o
              número da seqüência à qual
              corresponde r1 :
                  . se a seqüência s1j contém um nó
                                do tipo NILL com "l = k" :
                        . se k < s11 :
                                . inserir k no conjunto
                                  gt1
                        . se (k > sin1) e
                                 (s1j diferente de sin1) :
                                . inserir k no conjunto
                                  gf1
                  . inserir s1j na PrimeTree P1´
        . Localizar sin1 em Pi´
        . Trocar o nó do tipo NILL, com "l=sin1+1",
                                da sequência sin1 por "l=0".
```

Figura 3.5 : Algoritmo para construir $P_1´$ e a Tabela D_1

- Fase 2 - COBOL

As seqüências que compõem as novas **PrimeTrees** e a **Tabela** D_1, encontradas a partir de *P´* e **Ro** com a aplicação do algoritmo acima, são as seguintes :

---

2 O algoritmo utiliza os operadores do tipo PrimeTree relacionados na Seção 2.1.5.

Seqüências para a PrimeTree "tudo" :

```
11 :                                  l=12;
12 : q=0;                             l=13;
13 : p=0;                             l=14;
14 :                                  l=15;
15 : if (p==1)                        l=16
     else                             l=17;
16 :                                  l=12;
17 : if (q==1)                        l=18
      else                            l=19;
18 :                                  l=22;
19 :                                  l=20
20 : ++i;                             l=21;
21 :                                  l=22;
22 : if (q==1)                        l=23
     else                             l=24;
23 :                                  l=12;
24 :                                  l=25;
25 : display("i=",i);                 l=26;
26 :                                  l=0;
```

Seqüências para a PrimeTree "b" :

```
15 : if (p==1)                        l=16
     else                             l=17;
16 :                                  l=12;
17 : if (q==1)                        l=18
      else                            l=19;
18 :                                  l=22;
19 :                                  l=0;
```

Seqüências para a PrimeTree "c":

```
20 : ++i;                             l=21;
21 :                                  l=0;
```

Seqüências para a PrimeTree "d" :

```
22 : if (q==1)                        l=23
     else                             l=24;
23 :                                  l=12;
24 :                                  l=0;
```

Seqüências para a PrimeTree "e" :

```
25 : display("i=",i);                 l=26;
26 :                                  l=0;
```

Tabela 3.2 : Tabela de Desvios D_1

PrimeTree	início	fim	desvio-frente	desvio-trás
tudo	11	26	-	-
b	15	19	22	12
c	20	21	-	-
d	22	24	-	12
e	25	26	-	-

Como é possível observar na **Tabela** D_1, acima, as **PrimeTrees** *"tudo"*, *"c"* e *"e"* não possuem desvios para fora de suas **seqüências**; isto é, elas são **unidades** bem delineadas, ou ainda, **programas próprios**. No entanto, as **PrimeTrees** *"b"* e *"d"* possuem desvios para fora de seu bloco de **seqüências** ao mesmo tempo que são tratadas no programa original como **unidades de programa**.

Para que as **PrimeTrees** *"b"* e *"d"* sejam **programas próprios** elas precisam ser compostas com as **seqüências** para as quais ocorrem desvios, de forma que todo o caminho necessário de seus pontos de entrada a seus pontos de saída estejam presentes na mesma **PrimeTree**; esta recomposição é feita na **Fase 3**.

Vale observar que se em D_1 os conjuntos gt_1 e gf_1 estivessem vazios, para $i = 1,\ldots,n$, então o processo de normalização do código estaria completo.

Fase 3 : Tratamento de Desvios

Dado que existe uma **PrimeTree** P_1' para a qual gt_1, *ou* gf_1, contenha pelo menos um elemento, então P_1' será transformada em P_1''; de forma que P_1'' englobe todas as **seqüências** que tenham um caminho de gt_{1j}, $j = 1,\ldots,t_1$, até alguma **seqüência** entre s_{11} e s_{1n1} *ou* 0 - final do programa, e todas as **seqüências** de s_{1n1} até gf_{1j}, $j= 1,\ldots,f_1$, ou 0 - final do programa.

Na **PrimeTree** P_1'', o conjunto de **seqüências** que a compõem, $\{s_{11}'',\ldots,s_{1n1}''\}$, deve englobar todos os caminhos de execução; ou seja, todas as **seqüências** que ela possa executar. Esta premissa garante que P_1'' seja um **programa próprio**; ou seja, tenha um único ponto de entrada e um único ponto de saída e para cada nó exista um caminho do ponto de entrada ao último nó da **PrimeTree**.

As **seqüências** utilizadas para a recomposição das novas **PrimeTrees** serão copiadas da **PrimeTree** "**main**", pois nela se encontram todas as **seqüências** obtidas a partir do programa original.

O algoritmo para obter a transformação $P_1' \rightarrow P_1''$ é definido da seguinte forma :

```
Primeira Parte :
. Para i = 1,..., n ,
        onde n é o número de PrimeTrees em D1 :
    . Se, em  D1, gt1 ou gf1 não fôr vazio :
      . Criar uma nova PrimeTree P1"
      . para j = 1, n1; onde n1 é o número de
                          seqüências em P1´:
          . inserir a seqüência s1j em P1"
          . se (s1j não contém um nó do
                tipo OR) e
                (s1j contém um nó do tipo
                                NILL, com "l=c")
                . se ( c pertence a gt1) ou
                                (c pertence a gf1) :
                        CompoeCaminho(c)
      . Renomear P1" para P1´.
```

```
Segunda Parte : CompoeCaminho (c)
. Localiza a seqüência c
                na PrimeTree "main" => sc
. se ( c pertence a gt1 ) :
                remover c de gt1
. se ( c pertence a gf1 ) :
                remover c de gf1
. inserir sc em P1"
. se ( sc contém um nó do tipo NILL com "l=0") :
      . trocar o nó por um nó do tipo CALL
                com "exit()"
      . emitir uma mensagem de advertência
. se (sc contém um nó do tipo OR ) :
      . se o primeiro filho do OR tem um nó
          descendente do tipo NILL com "l=k"
          . se a seqüência de número k
            não é localizada em P1´ou P1"
                        CompoeCaminho(k)
      . se o segundo filho do OR tem um nó
           descendente do tipo NILL com "l=k"
           . se a seqüência de número k
                  não é localizada em P1´ou P1"
                        CompoeCaminho(k)
. se não :
      . se sc contém um nó do tipo NILL com
                                        "l=k"
          . se a seqüência de número k
                  não é localizada em P1´ou P1"
                                CompoeCaminho(k)
```

Figura 3.6 : Algoritmo para Tratamento de Desvios - Fase 3 - COBOL

O algoritmo acima é dividido em duas partes. A primeira parte encarrega-se de copiar todas as seqüências de P1´ para P1" e, sempre que encontrar um desvio, em uma dessas seqüências, fazer uma chamada para a segunda parte do algoritmo. A segunda parte compõe

recursivamente[3] as **seqüências** de **P_1"** com as **seqüências** da **PrimeTree "main"**, completando o caminho dos desvios. A recursão termina quando a próxima **seqüência** a ser executada pertence à **PrimeTree P_1"** ou à **P_1´**; pois a **seqüência** pode não ter sido copiada para **P_1"** e pertencer a **P_1´**.

A seguir são ilustradas as **seqüências** que compõem as novas **PrimeTrees** *"d"* e *"b"*; obtidas com a aplicação do algoritmo anterior :

Seqüências para a PrimeTree "b" :

```
15 : if (p==1)          l=16
     else               l=17;
16 :                    l=12;
12 : q=0;               l=13;
13 : p=0;               l=14;
14 :                    l=15;
17 : if (q==1)          l=18
     else               l=19
18 :                    l=22;
22 : if (q==1)          l=23
     else               l=24;
23 :                    l=12;
24 :                    l=25;
25 : display("i=",i);   l=26;
26 : exit();            l=0;
19 :                    l=0;
```

Seqüências para a PrimeTree "d" :

```
22 : if (q==1)    l=23
     else         l=24;
23 :              l=12;
12 : q=0;         l=13;
13 : p=0;         l=14;
14 :              l=15;
15 : if (p==1)    l=16
     else         l=17;
16 :              l=12;
17 : if(q==1)     l=18
     else         l=19;
18 :              l=22;
19 :              l=20;
20 : ++i;         l=21;
21 :              l=22;
24 :              l=0;
```

Como pode ser observado, todos os caminhos possíveis, ou seja, **seqüências** que as **PrimeTrees** englobam, estão presentes em cada **PrimeTree**.

Deve-se notar, que a nova **PrimeTree "b"** (ou **unidade "b"**) pode interromper a execução do programa; o desvio que leva suas **seqüências** para o fim do programa foi trocado pela função **"exit()"** (vide

3 Deve-se observar que a segunda parte do algoritmo é recursiva e, a nível de implementação, a seqüência s_c deve ser uma variável local; para que seu valor seja conservado caso contenha um nó do tipo OR.

seqüenciamentos do **Apêndice A**). Nesse caso, o processo da **Fase 3** limita-se a emitir uma mensagem de advertência, pois pressupõe-se que o programa original está em produção, conseqüentemente a interrupção de sua execução na **unidade "b"** deve corresponder a alguma especificação.

3.2 Esquema de Decomposição para a Linguagem FORTRAN

FORTRAN foi a primeira linguagem de programação de alto-nível, suas primeiras versões datam de meados da década de 50 [PRATT, 84]. Nessa época o grande objetivo era obter eficiência de execução para os programas após suas compilações, devido ao uso de linguagens de montagem (ASSEMBLY) para o desenvolvimento de sistemas de software e a grande eficiência desses sistemas.

A linguagem de programação **FORTRAN** herdou de sua origem o seu primeiro objetivo, que é a eficiência do código gerado após compilação; por este motivo possui estruturas de controle muito simples e não muito elegantes. As estruturas de controle incluem "**goto**", comandos rotulados, uma estrutura de seleção simples com "**if-then-else**", expressões escritas como fórmulas matemáticas e um único tipo de comando de iteração [PRATT, 84].

O único tipo de abstração que a linguagem provê é o uso de três tipos de subprogramas. O primeiro tipo é denominado função ("**function**"); retorna um único valor correspondente ao cálculo da função modelada pelo subprograma. O segundo tipo é denominado subrotina ("**subroutine**"); a lista de parâmetros de uma subrotina pode ter seus valores recomputados no corpo do subprograma. Nestes dois tipos, os resultados de valores computados ao longo do subprograma podem também estar armazenados em blocos comuns ("**common-blocks**"), cuja funcionalidade é semelhante ao uso de variáveis globais [HEHL, 79]. O terceiro tipo é denominado "**statement function**", pois consiste de uma única linha de comando; é declarado no início de um subprograma e/ou programa principal e é local à unidade [PRATT, 84].

Uma característica dos subprogramas dos tipos "**function**" e "**subroutine**" é a redeclaração de seus pontos de entrada. Normalmente um subprograma inicia a execução na sua primeira linha de código executável e termina quando um comando "**return**" é encontrado. No entanto, nestes subprogramas pode-se declarar mais um ponto de entrada com o uso do comando "**ENTRY**" [HEHL, 79].

A simplicidade das estruturas de controle da linguagem facilita o processo de obtenção dos **seqüenciamentos**; relacionados no **Apêndice B**. A existência de subprogramas devidamente declarados e com escopo bem delimitado também facilita a identificação das **unidades de programa**.

A existência de mais de um ponto de entrada em um subprograma, indica que este outro ponto de entrada ("**entry-point**") delimita uma **nova unidade de programa**; e como os "**entry-points**" fazem parte de uma

unidade maior podem ocorrer desvios entre essas **unidades**. Para melhor visualizar o problema, consideremos o seguinte exemplo :

```
      do 40 p = 0 , 1
      do 30 q = 0, 1
            call b(p,q)
            call d(p,q)
            call tudo(p,q)
30          continue
40    continue
      stop
      end

      subroutine tudo (p,q)
      rb = 0
      rd = 0
12    q = 0
      p = 0
      goto 24
      entry b(p,q)
      rb = 1
      rd = 0
24    if (p.eq.1)     goto 12
      if (q.eq.1)     goto 30
      if (rb.eq.1)    return
30    i = i + 1
      goto 36
      entry d(p,q)
      rd = 1
      rb = 0
36    if (q.eq.1)     goto 12
      if (rd.eq.1)    return
      write (0,50) i
50    FORMAT(I2)
      return
      end
```

Deve-se notar que no exemplo acima a subrotina "**tudo**" contém duas subdivisões, representadas pelos "**entry-points**" "**b**" e "**d**". De fato, tanto "**b**" como "**d**" são também subrotinas, logo deverão ser geradas **PrimeTrees** para cada uma delas. Além disso, como a subrotina "**tudo**" engloba os comandos referentes a "**b**" e "**d**", desvios do escopo de "**b**" e "**d**" para outros comandos em "**tudo**" podem ocorrer, como ilustrado no exemplo. Os desvios acarretam entrelaçamentos entre estas subrotinas que, necessariamente, devem ser tratados pelo **esquema de decomposição**.

A estratégia utilizada é muito similar a do esquema de decomposição para códigos fonte na linguagem de programação **COBOL**. No exemplo anterior existem entrelaçamentos entre as subfunções dentro de uma mesma função, cujo tratamento é semelhante ao dos entrelaçamentos de "**perform´s**" e "**go´s**" (vide Seção 3.1). O **esquema de decomposição** para códigos implementados em **FORTRAN** também possui três fases.

A primeira fase consiste no **seqüenciamento** de cada subprograma e se e somente se houver "**entry-points**" as outras fases serão

necessárias. Como o escopo de subprogramas é bem delimitado, não existem desvios para fora desse escopo, o que justifica a não aplicação das outras fases caso não existam "**entry-points**". A segunda fase é a geração de **PrimeTrees** para cada "**entry-point**" e tabulação de desvios. Se houver desvios na segunda fase, então a terceira fase, que é o tratamento de desvios, é aplicada às **PrimeTrees** geradas.

Fase 1 : Seqüenciamento dos Subprogramas do Programa Original e Tabulação de Seus "Entry-Points"

Dado um programa **P**, com seu código fonte na linguagem de programação **FORTRAN**, cada um de seus subprogramas P_i será transformado em $P_i' = \{s_{i1}, s_{i2}, \ldots, s_{ini}\}$; onde :

P_i' : uma **PrimeTree**, composta por todas as **seqüências** geradas para as linhas de código de P_i

s_{ij} : **seqüência** resultante do **seqüenciamento** de alguma linha de código de P_i, para $j = 1,..,n_i$; onde n_i é o número de **seqüências** em P_i'. O **seqüenciamento** é obtido de acordo com as proposições do **Apêndice B**.

i : varia de **1** a **n**; onde **n** é o número de subprogramas de **P**.

O **seqüenciamento** dos subprogramas originais, transformação $P_i \rightarrow P_i'$, independe da existência de mais de um ponto de entrada em P_i; ou seja, o **seqüenciamento** independe da existência de comandos do tipo "**ENTRY**" em seu escopo.

Para que possam ser identificadas **novas unidades de programa** a partir de um certo P_i', paralelamente à transformação $P_i \rightarrow P_i'$, são montadas tabelas de "**entry-points**" para cada P_i' que os possua, denominadas de E_{io}, e definidas do seguinte modo :

E_{io} : Tabela de "Entry-Points" para P_i'

"entry-point"	início	fim
e_1	s_{11}	s_{1f}
e_2	s_{21}	s_{2f}
...	...	...
e_{nk}	s_{nk1}	s_{nkf}

Onde,

e_k : nome do *k-ésimo* "**entry-point**" da **PrimeTree** P_i'

s_{k1} : número da **seqüência** inicial do *k-ésimo* "**entry-point**" da **PrimeTree** P_i'

s_{kf} : número da **seqüência** final do *k-ésimo* "**entry-point**" da **PrimeTree** P_i', correspondente à decomposição do primeiro comando "**return**" encontrado após o "**entry-point**"

k : varia de *1 a* n_k; onde n_k é o número de "entry-points" em P_1'.

Por exemplo, para cada subprograma do programa exemplificado no início da seção tem-se os seguintes **seqüenciamentos**[4] :

Seqüências para a PrimeTree "main" :

```
1  : p=0;                    l=2;
2  : q=0;                    l=3;
3  : b(p,q);                 l=4;
4  : d(p,q);                 l=5;
5  : tudo(p,q);              l=6;
6  : ++q;                    l=7;
7  : if (q<=1)               l=3
     else                    l=8;
8  : ++q;                    l=9;
9  : if (p<=1)               l=2
     else                    l=10;
10 :                         l=0;
```

Seqüências para a PrimeTree "tudo":

```
1  : rb=0;                   l=2;
2  : rd=0;                   l=3;
3  : p=0;                    l=4;
4  : q=0;                    l=5;
5  :                         l=8;
6  : rb=1;                   l=7;
7  : rd=0;                   l=8;
8  : if (p==1)               l=9
     else                    l=10;
9  :                         l=3;
10 : if (q==1)               l=11
     else                    l=12;
11 :                         l=18;
12 : if (rb==1)              l=13
     else                    l=14;
13 :                         l=0;
14 : ++i;                    l=15;
15 :                         l=18;
16 : rd=1;                   l=17;
17 : rb=0;                   l=18;
18 : if (q==1)               l=19
     else                    l=20;
19 :                         l=3;
20 : if (rd==1)              l=21
     else                    l=22;
21 :                         l=0;
22 :write(i,formato,unidade);
                             l=23;
23 :                         l=0;
```

---

4 *Os seqüenciamentos são obtidos de acordo com as sugestões do Apêndice B.*

Deve-se notar que os **seqüenciamentos** anteriores geram respectivamente as **PrimeTrees** "**main**" e "**tudo**". Paralelamente ao **seqüenciamento** é gerada a **Tabela de "entry-points"** E_{20}, correspondente ao subprograma "**tudo**", a seguir :

Tabela 3.3 : Tabela de "Entry-Points" para P_2' - E_{20}

"entry-point"	início	fim
b	6	13
d	16	21

Se, para o programa do exemplo, existissem mais subprogramas com "**entry points**" teriam sido geradas outras tabelas E_{i0} (i diferente de 2). E, ainda, se para os subprogramas do exemplo não existisse um único "**entry point**", ou seja, todas as tabelas E_{i0} estivessem vazias, o [illegible] do código estaria encerrado; nesse caso, cada **PrimeTree** obtida seria a **Primeira Forma Normal** desses subprogramas.

Fase 2 : Divisão das Unidades e Tabulação de Desvios

Para cada **PrimeTree** P_i', cuja tabela E_{i0} não esteja vazia, serão geradas novas **PrimeTrees** P_{ik}"= $\{s_{k1},\ldots,s_{km}\}$; onde $k = 1,\ldots,n_k$, n_k é o número de "**entry-points**" no subprograma P_i e **m** é o número de **seqüências** de P_{ik}".

Dentro do conjunto de **seqüências** de uma **PrimeTree** P_i', que contenha "**entry-points**", podem ocorrer desvios entre as sub-funções definidas; as sub-funções geram novas **unidades de programa** correspondentes à P_{ik}". Entende-se por **desvio** a existência de um nó do tipo **NILL** entre as **seqüências** de P_{ik}", s_{k1} a s_{km}, com um conteúdo "**l=d**" tal que :

. $0 < d < s_{k1}$, então existe um desvio para uma **seqüência** antecessora à primeira **seqüência** de P_{ik}".

. $d > s_{km}$, então existe um desvio para uma **seqüência** sucessora à última **seqüência** de P_{ik}".

Assim, paralelamente à geração de novas **PrimeTrees**, para cada tabela E_{i0} será construída uma tabela de desvios D_{i1}, definida do seguinte modo :

D_{i1} : Tabela de Desvios para cada PrimeTree P_i'

PrimeTree	início	fim	desvio-frente	desvio-trás
P_{i1}'	s_{i1}	s_{im1}	$gf_{i1},\ldots,gf_{ir1}$	$gt_{i1},\ldots,gt_{it1}$
P_{i2}'	s_{21}	s_{2m2}	$gf_{21},\ldots,gf_{2r2}$	$gt_{21},\ldots,gt_{2t2}$
...	...	...	...	...
P_{ink}'	s_{nk1}	s_{nkmk}	$gf_{nk1},\ldots,gf_{nkrk}$	$gt_{nk1},\ldots,gt_{nktk}$

Onde,

P_{1k}' : nome da *k-ésima* **PrimeTree** gerada a partir de P_1', $k = 1,\ldots,n_k$, e n_k é o número de linhas em E_{1o}

s_{k1} : número da seqüência inicial da **PrimeTree** P_{1k}', que é o mesmo de s_{k1} em E_{1o}

s_{knk} : número da seqüência final da **PrimeTree** P_{1k}', que é o mesmo de s_{kf} em E_{1o}

gf_k : conjunto de desvios para seqüências antecessoras às seqüências de P_{1k}'. Ou ainda, $gf_{kt} < s_{k1}$, para $t=1,\ldots,f_k$, sendo f_k o tamanho do conjunto gf_k de desvios para frente.

gt_k : conjunto de desvios para seqüências sucessoras às seqüências de P_{1k}'. Ou ainda, $gt_{kt} > s_{km}$, para $t=1,\ldots,t_k$, sendo t_k o tamanho do conjunto gf_k de desvios para trás.

Vale observar que P_1', a **PrimeTree** resultante da **Fase 1**, não contém desvios do tipo de gf_k e gt_k, pois contém todas as seqüências do subprograma original, por esse motivo é mantida intacta no processo de transformação.

A construção das novas **PrimeTrees** P_{1k}', i = 1,...,n e k = 1,...,n_k, e a tabela D_{11}, utilizando os operadores do tipo **PrimeTree**, é obtida com o seguinte algoritmo :

```
. Para i = 1,..,n ; onde n é o número de subprogramas de P
   . Se E1o não é vazia :
      . Criar a Tabela D11
      . para k = 1, ..., nk; onde nk é o número de
                                   "entry-points" de E1o :
         . Criar a PrimeTree P1k' => PT_Cria(ek)
         . Copiar para a tabela D11 as informações do
           nome do "entry-point" ek como o nome de
           P1k',sk1 e skf como o número das
           seqüências inicial (sk1)  e final (skm) de P1k'
         . Localizar sk1 em P1' -> guardar em s
         . Faça :
            . se a seqüência s contém um nó do tipo
                                          NILL com "l = d" :
               . se 0 < d < sk1 ->
                                  inserir d no conjunto gtk
               . se d > skf ->
                                  inserir d no conjunto gfk
               . se d = skf ->
                                 trocar o conteúdo do nó do
                                 tipo NILL por "l = 0"
            . s <- Próxima seqüência de P1'
         . enquanto ( s <= skf )
```

Figura 3.7 : Algoritmo para obter P_{1k}' e a Tabela D_{11}

- Fase 2 - FORTRAN

Utilizando o algoritmo anterior pode-se gerar as novas **PrimeTrees** construídas a partir de P_2' e E_{20} (obtidas na fase anterior) e a **Tabela** D_{21}, que são as seguintes :

Seqüências para a PrimeTree "b":

```
6  : rb=1;              l=7;
7  : rd=0;              l=8;
8  : if (p==1)          l=9
     else               l=10;
9  :                    l=3;
10 : if (q==1)          l=11
     else               l=12;
11 :                    l=18;
12 : if (rb==1)         l=13
     else               l=0;
13 :                    l=0;
```

Seqüências para a PrimeTree "d":

```
16 : rd=1;              l=17;
17 : rb=0;              l=18;
18 : if (q==1)          l=19
     else               l=20;
19 :                    l=3;
20 : if (rd==1)         l=21
     else               l=0;
21 :                    l=0;
```

Tabela 3.4 : Tabela de Desvios para P_{2k}' - D_{21}

PrimeTree	início	fim	desvio-frente	desvio-trás
b	6	13	18	3
d	16	21	-	3

Como é possível observar em D_{21} as **PrimeTrees** "b" e "d" possuem desvios para fora de suas **seqüências**; isto é, não são **programas próprios**, mesmo no programa original P.

Para que as **PrimeTrees** "b" e "d" sejam **programas próprios** elas precisam ser compostas com as **seqüências** para as quais ocorrem desvios, de forma que todo o caminho necessário de seus pontos de entrada a seus pontos de saída estejam completos na mesma **PrimeTree**; isto é feito na próxima fase.

Vale observar que se em D_{11} os conjuntos gt_k e gf_k estivessem vazios, para $i = 1,\ldots,n$ e $k = 1,\ldots,n_k$, então o processo de normalização do código estaria completo.

Fase 3 : Tratamento de Desvios

Dado que existe uma **PrimeTree** P_1' e uma tabela D_{11}, e em D_{11} os conjuntos gt_k ou **gfk**, para $k = 1,..,nk$, contenham pelo menos um elemento, então existe uma **PrimeTree** P_{1k}' que não é um **programa próprio**.

Se esta premissa é verdadeira então P_{1k}' deve ser transformada em P_{1k}''. P_{1k}'' deve englobar todas as **seqüências** de P_{1k}' e mais as **seqüências** para as quais os desvios formam novos caminhos de execução.

O algoritmo para obter a transformação P_{1k}' -> P_{1k}'' é dividido em duas partes, pois deve copiar todas as **seqüências** de P_{1k}' em P_{1k}'', e sempre que houver um desvio em uma **seqüência** s_{kt}, para $t = 1,...,m$, o caminho será composto com as **seqüências** copiadas de P_1' até retornar a alguma **seqüência** de P_{1k}' ou a **seqüência** pertencer à nova **PrimeTree** P_{1k}''. O algoritmo é ilustrado na Figura 3.8.

A aplicação do algoritmo aos resultados obtidos na fase anterior obtém as **seqüências** que formam as novas **PrimeTrees** P_{21}'' ("b") e P_{22}'' ("d"), que são :

Seqüências para PrimeTree "b":

```
6  : rb=1;                    l=7;
7  : rd=0;                    l=8;
8  : if (p==1)                l=9
     else                     l=10;
9  :                          l=3;
3  : p=0;                     l=4;
4  : q=0;                     l=5;
5  :                          l=8;
10 : if (q==1)                l=11
     else                     l=12;
11 :                          l=18;
18 : if (q==1)                l=19
     else                     l=20;
19 :                          l=3;
20 : if (rd==1)               l=21
     else                     l=22;
21 :                          l=0;
22 : write(i,formato,unidade);
                              l=23;
23 :                          l=0;
12 : if (rb==1)               l=13
     else                     l=0;
13 :                          l=0;
```

```
Parte 1 : Copiar  todas as seqüências de P_ik' para P_ik"
.Para i = 1,..., n; onde n é o num. de subprogramas de P:
   . Se existe a tabela D_i1 :
      . Para k = 1,...,n_k; onde n_k é o número de linhas em D_i1:
         . se ( gf_k não é um conjunto vazio) ou
                                  (gt_k não é um conjunto vazio ) :
            . Criar a PrimeTree P_ik"
            . Para t = 1,...,m,
                onde m é o número de seqüências em P_ik':
                . Inserir s_kt, de P_ik', em P_ik"
                . se (s_kt não contém nós do tipo OR) e
                     (skt contém um nó do tipo NILL,com "l=c")
                     . se (c pertence a gt_k) ou
                          (c pertence a gf_k)
                                -> CompoeCaminho(c)
            . Renomear P_ik" para P_ik'
```

```
Parte 2 :  CompoeCaminho (c)
. Localiza a seqüência c em P_i' -> guardar em s_c
. se ( c pertence a gt_k) ->  retirar c de gt_k
. se (c pertence a gf_k ) ->  retirar c de gf_k
. inserir a seqüência sc em P_ik"
. se (s_c contém um nó do tipo OR ) ->
      . se o primeiro filho do nó do tipo OR
                          contém um nó do tipo NILL com "l=d"
           . se a seqüência de número d
                          não pertence à P_ik" e
                          não pertence a P_ik'
                                -> CompoeCaminho(d)
      . se o segundo filho do nó do tipo OR
                         contém um nó do tipo NILL com "l=d"
           . se a seqüência de número d não
                               pertence à P_ik" e
                               não pertence a P_ik'
                                     -> CompoeCaminho(d)
. se (s_c contém um nó do tipo NILL com "l=d") ->
      .  se a seqüência de número d não pertence
                                  à P_ik" e não pertence a P_ik'
                                     -> CompoeCaminho(d)
```

Figura 3.8 : Algoritmo para Tratamento de Desvios - Fase 3 - FORTRAN

Seqüências para a PrimeTree "d":

```
16 : rd=1;                  l=17;
17 : rb=0;                  l=18
18 : if (q==1)              l=19
     else                   l=20;
19 :                        l=3;
3  : p=0;                   l=4;
4  : q=0;                   l=5;
5  :                        l=8;
8  : if (p==1)              l=9
     else                   l=10;
9  :                        l=3;
10 : if (q==1)              l=11
     else                   l=12;
11 :                        l=18;
12 : if (rb==1)             l=13
     else                   l=14;
13 :                        l=0;
14 : ++i;                   l=15;
15 :                        l=18;
20 : if (rd==1)             l=21
     else                   l=0;
21 :                        l=0;
```

Deve-se notar que as duas novas **PrimeTrees** agora são **programas próprios**, ou **unidades de programa completas**, pois todos os caminhos (**seqüências**) envolvidos em suas execuções pertencem à mesma **PrimeTree**.

3.3. Esquema de Decomposição para a Linguagem C

Programas escritos na linguagem de programação **C** normalmente são compostos por vários subprogramas, cujas referências são feitas por chamadas explícitas [KERNIGHAN, 86]. Os subprogramas são sempre **funções**; podem retornar um valor de dado de um tipo qualquer ou nenhum valor e serem parametrizados ou não. A forma de retorno de valores computados por subprogramas não ocorre exclusivamente através de parametrização, podendo ser usadas variáveis de escopo global.

As estruturas de controle incluem seleções simples e múltiplas, iterações com laços bem definidos, agrupamentos de comandos e desvios. Os desvios podem ser do tipo "**goto**", quebra de laços ("**break**" - desvios para fora de laço ou bloco de comandos) e desvio para a próxima iteração de um laço ("**continue**") [KERNIGHAN, 86].

Como as estruturas de controle da linguagem intermediária adotada são um subconjunto da linguagem C, as transformações que levam ao

seqüenciamento de um programa são simples, como pode ser observado no **Apêndice C.**

O escopo de funções é bem delimitado, não podendo ocorrer desvios entre as **unidades de programa**; logo o **esquema de decomposição** consiste de uma fase única, que corresponde ao **seqüenciamento** dos subprogramas. Os **seqüenciamentos** são necessários e suficientes para obter um programa C em sua **Primeira Forma Normal.**

Uma função tem um único ponto de entrada, mas pode conter mais de um comando "**return**"; o que corresponde a ter mais de um ponto de saída. Este problema é facilmente contornável pelo **seqüenciamento** do comando "**return**", como mostrado no início deste capítulo. Este resultado permite considerar uma função como uma **unidade de programa** que é um **programa próprio.**

Dado um programa P, com seu código fonte na linguagem C, para o qual cada função é denominada P_i; a fase única do **esquema de decomposição** consiste em transformar cada P_i em $P_i' = \{s_1, s_2, \ldots, s_{ni}\}$; onde :

P_i' : uma única **PrimeTree**, composta por todas as **seqüências** de P_i.

s_i : **seqüência** resultante do **seqüenciamento** de alguma linha de código de P_i, para i = 1,..,n. O **seqüenciamento** é obtido de acordo com a proposta do **Apêndice C.**

n_i : número de **seqüências** em P_i'.

Para ilustrar o **esquema de decomposição** será utilizado o seguinte código fonte; que é composto por duas funções ("main" e "exemplo") :

```
main ()
{
     int  fem[50],male[50];

     extern void leia();
     extern void exemplo();

     leia(fem,male);
     exemplo(fem,male);
}
```

```
void        exemplo (fem, male)
int         fem[50],
            male[50];
{
      int k,i;

            k = 0;
quatro:     ++k;
            if ( k > 50 )
                        goto nove;
            i = 1;
            do {
                  if ( fem[i] )
                        goto sete;
                  if ( ! male[i][k] )
                              break;
                  goto quatro;
sete:             if ( !male[i][k] )
                        goto quatro;
                  ++i;
            } while ( i < 8 ) ;
            printf ( "k=[%d]\n",k);
            goto quatro;
nove :
            ; /*nada */
}
```

As seqüências[5] que originam as PrimeTrees, correspondentes às Primeiras Formas Normais das duas funções do exemplo, são :

Seqüências para a PrimeTree "main"

```
1 : leia(fem,male);              1=2;
2 : exemplo(fem,male);           1=3;
3 :                              1=0;
```

---

5 *Os seqüenciamentos são obtidos de acordo com as sugestões do Apêndice C.*

Seqüências para a PrimeTree "exemplo":

```
1  : k=0;                    l=2;
2  : ++k;                    l=3;
3  : if (k>50)               l=4
     else                    l=5;
4  :                         l=17;
5  : i=1;                    l=6;
6  : if (fem[i])             l=7
     else                    l=8;
7  :                         l=11;
8  : if (!male[i][k])        l=9
     else                    l=10;
9  :                         l=15;
10 :                         l=2;
11 : if (!male[i][k])        l=12
     else                    l=13;
12 :                         l=2;
13 : ++i;                    l=14;
14 : if (i<8)                l=6
     else                    l=15;
15 : printf(...);            l=16;
16 :                         l=2;
17 :                         l=0;
```

Deve-se notar que para um programa escrito na linguagem **C**, dado que são disponíveis os códigos fontes de suas funções, é suficiente **seqüenciar** cada função para obter a **Primeira Forma Normal** de todo o programa. Não há a necessidade de subdivisões das **PrimeTrees** obtidas, como no caso dos códigos fonte em linguagem **COBOL** e **FORTRAN**, pois para **C** não existe desvio para fora do corpo de uma função ou sobreposições funcionais.

4. REDUÇAO DA ESTRUTURA DO PROGRAMA

Obter a **Primeira Forma Normal** do código fonte de um produto de software é produzir um novo código padronizado através de estruturas elementares, cuja equivalência funcional com o código original é mantida por uma única iteração, seleções simples, atribuições e testes sobre um único contador adicional.

Essa forma não é adequada para servir de base a ferramentas de reengenharia que usem análise de interdependência de variáveis [REZENDE, 92]. As **seqüências** da **Primeira Forma Normal** são fortemente acopladas pelo uso do contador adicional, o que inviabiliza obter partições de um código nesta forma [MACARIO, 92].

Ainda, se considerarmos o número de ciclos de um programa como medida de complexidade de um código [McCABE, 76], a **Primeira Forma Normal** de um código possui tantos ciclos quantas são as funções (linhas de código), mais um (a única iteração). Logo, utilizando esta métrica a representação do código nessa forma é altamente complexa, além de não ser suficientemente clara e eficiente [LINGER, 79].

O **Modelo de Redução** reduz o uso do contador adicional diminuindo o número de ciclos do programa. O modelo é baseado na propriedade de composição de programas primos, recursividade de programas e em uma análise semântica do contador adicional [LINGER, 79].

A Figura 4.1 ilustra as etapas do **Modelo de Redução**. A etapa de "**Composição**" corresponde à simplificação da **Primeira Forma Normal** proposta por Linger e Mills (vide Seção 2.1.3), exceto por não permitir a replicação de partes do programa; esta etapa realiza as composições de programas primos que resultam nas composições das **seqüências** presentes na **Primeira Forma Normal** do código. Para reduzir o uso do contador adicional durante o processo de composição, foi incluído no modelo original de Linger e Mills um algoritmo de "**Fatoração**", Etapa 2 do modelo, que age sobre cada **seqüência** composta. A Etapa 3 do modelo corresponde a um processo de identificação de iterações ("**Identifica - Loops**"), que foi acoplado ao modelo original, também para reduzir o uso do contador adicional e melhorar a legibilidade do código estruturado final. As etapas são executadas seqüencialmente; no entanto, o processo associado ao **Modelo de Redução** é considerado completo quando não é mais possível realizar composições das **seqüências** da **PrimeTree** e não existem mais "**loops**" no programa.

Aplicando o **Modelo de Redução** à **Primeira Forma Normal** de um código fonte qualquer, todo **código morto** presente no código original é eliminado. **Código morto** é um trecho de código inatingível através do fluxo de execução do programa [MILLER, 91]. Na maior parte dos casos, a presença de código morto em um programa ocorre devido à evolução contínua de programas [LEHMAN, 80]; quando um código passa por vários processos de manutenção / evolução comumente sofre uma piora estrutural acentuada a cada alteração, o que pode levá-lo a apresentar partes inutilizáveis.

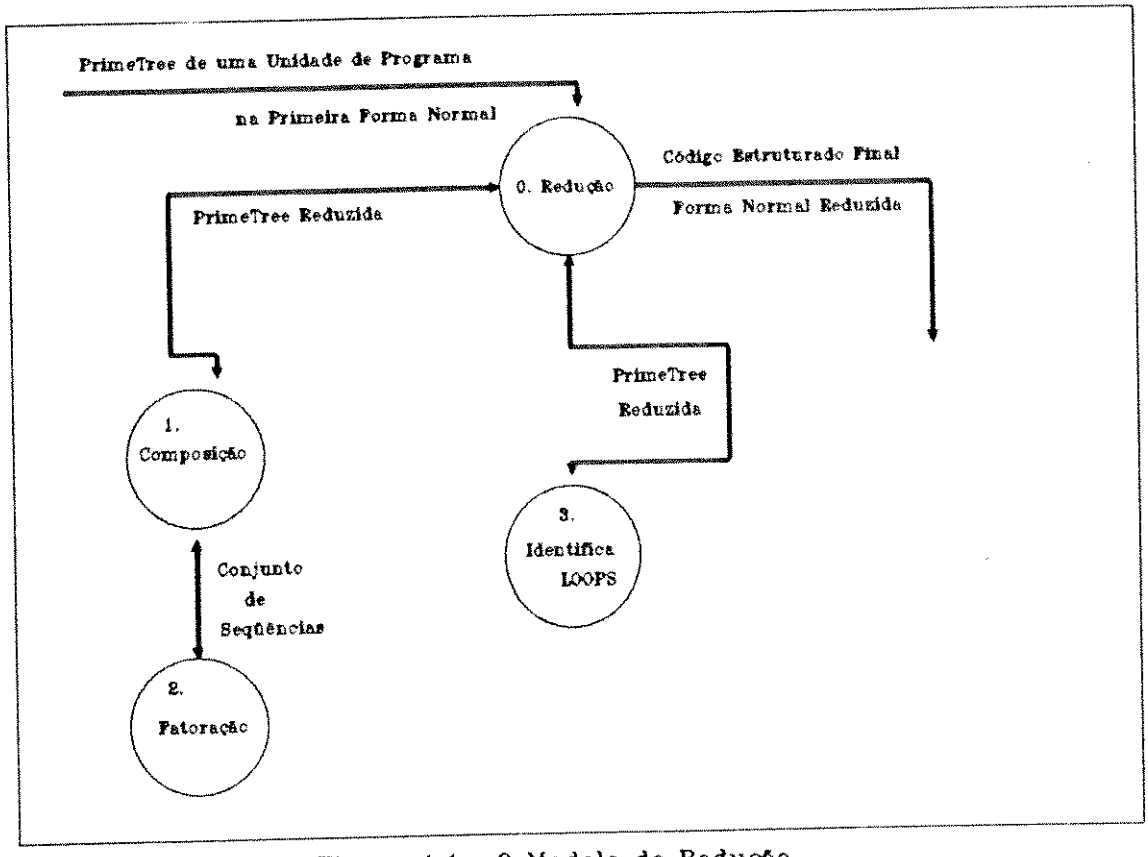

Figura 4.1 : O Modelo de Redução

O **Modelo de Redução** obtém uma nova representação para o programa denominada **Forma Normal Reduzida**, também padronizada pelo conjunto básico de estruturas de controle - comandos seqüenciais, **"do-while"** e **"if-then-else"**. Esta forma é funcionalmente equivalente à **Primeira Forma Normal** e, conseqüentemente, funcionalmente equivalente ao código original.

A **Forma Normal Reduzida** de um código fonte qualquer, obtida a partir de sua **Primeira Forma Normal**, não conterá o contador adicional se e somente se o código original não contém blocos de iteração entrelaçados por desvios, ou blocos de iteração e/ou seleção com mais de um ponto de saída construídos através de desvios. Caso contrário, a forma de representação do código estruturado final, resultante da aplicação do **Modelo de Estruturação**, mantém o contador adicional.

Na próxima seção é detalhado o **Modelo de Redução** e na Seção 4.2 é dado um exemplo da aplicação do modelo e o resultado correspondente.

4.1. O Modelo de Redução

A **Primeira Forma Normal** de um programa é um conjunto de **seqüências**, que são programas próprios formados por programas primos.

Vimos na Seção 2.1.1 que se um nó funcional de um programa primo é trocado por um outro programa primo, a operação resulta em um novo programa próprio denominado **programa composto.**

A idéia da composição de programas sobre a **Primeira Forma Normal** de um código consiste em trocar qualquer nó funcional que corresponda a uma atribuição do tipo *"I=J"* pela **seqüência** s_J [LINGER, 79], para $j > 0$ e $j <= n$, onde n é o número de **seqüências** na **PrimeTree**. Ou ainda, trocar qualquer nó do tipo **NILL** com conteúdo *"I=J"* de uma **seqüência** s_I, *i diferente de j*, pelos nós funcionais de qualquer tipo da **seqüência** s_J, aos quais denominamos "**corpo da seqüência**". Ao realizar a troca, a **seqüência** s_J é eliminada da **PrimeTree**; a Figura 4.2 ilustra este processo.

Deve-se notar que a composição de s_I com s_J, Figura 4.2, resulta em uma nova **seqüência** s_I, à qual foi acoplada a funcionalidade de s_J. Como s_J é eliminada, o número de **seqüências** da **PrimeTree** diminui, passando de n **seqüências** iniciais para $n-1$, isto é, o número de ciclos [McCABE, 76] é reduzido em uma unidade a cada composição.

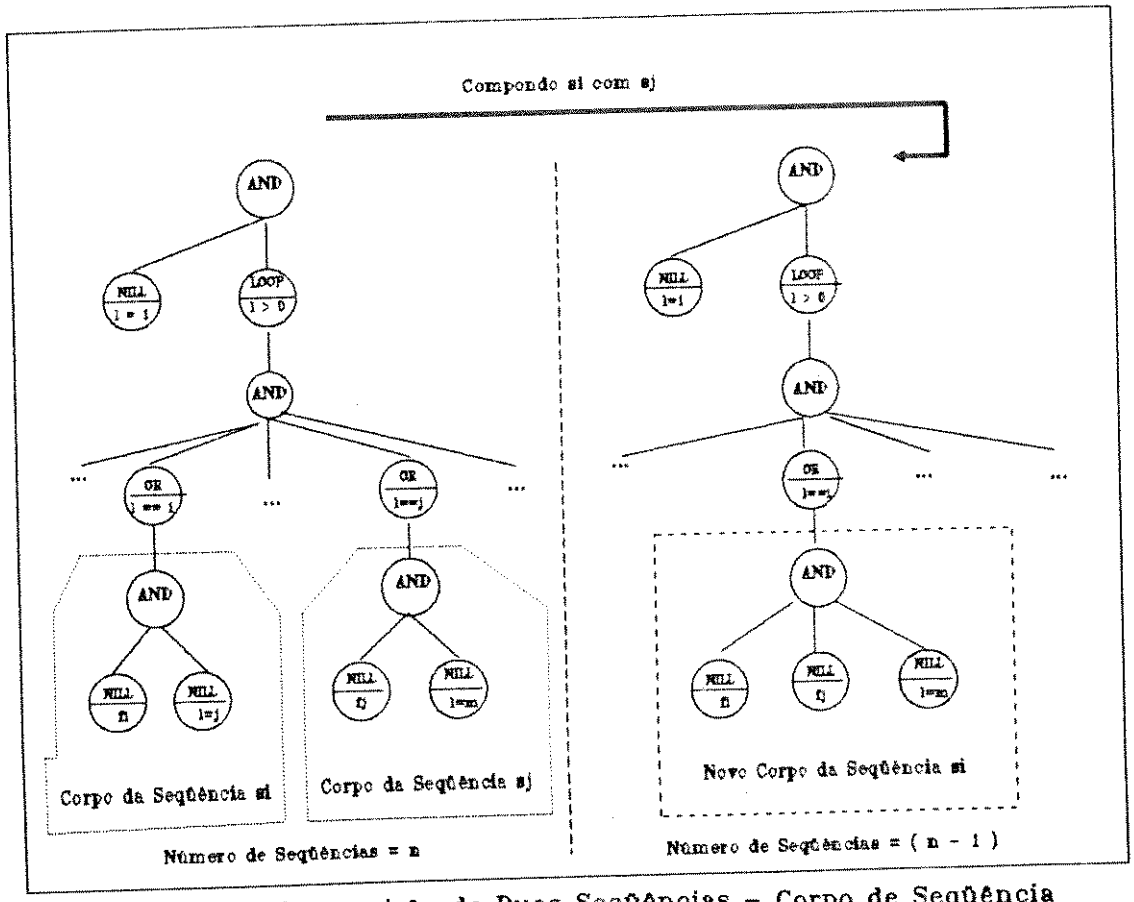

Figura 4.2 : Composição de Duas Seqüências - Corpo de Seqüência

A existência de uma atribuição do tipo *"1=j"* em uma **seqüência** s_j implica na presença de recursividade em s_j, ou seja, s_j é algum tipo de iteração. Logo, se isto ocorre, as composições de s_j não podem ser feitas pois, ao substituir a atribuição *"1=j"* por s_j, a mesma seria reincluída e a **seqüência** não poderia ser eliminada.

Se para uma **seqüência** s_j, $j > 0$, não existir pelo menos uma atribuição *"1=j"* em uma **seqüência** s_i, *i diferente de j*, então a **seqüência** s_j é **código morto**. Se a atribuição não existe, ou só existe em s_j, então não existe um caminho que inclui s_j no fluxo de execução do programa [LINGER, 79].

Se a composição de s_1 com s_j for realizada para qualquer atribuição *"1=j"*, $0 < j <= n$ e *j diferente de 1*, muitos trechos de código serão replicados. Para que isto não ocorra, no **Modelo de Redução** a aplicação da composição restringe-se aos casos nos quais, na **PrimeTree**, exista um único nó do tipo **NILL** com conteúdo *"1=j"*.

Uma forma de reduzir o número de nós do tipo **NILL** correspondentes a atribuições do tipo *"1=j"* na **PrimeTree**, sem realizar composições e mantendo a equivalência funcional, consiste em simplificar as **saídas** de um nó do tipo **OR**. Denominamos por **saída** a todo nó do tipo **NILL** que contém uma atribuição do tipo *"1=j"*, $0<=j<=n$, e é filho de um nó do tipo **OR**. A Figura 4.3 ilustra um nó do tipo **OR** para o qual existem duas saídas iguais, isto é, dois nós do tipo **NILL** com conteúdos correspondentes a atribuições do tipo *"1=j"*. Na figura, se as saídas forem eliminadas e, a seguir for inserida uma saída de igual conteúdo como irmã sucessora do nó do tipo **OR**, tem-se uma representação funcionalmente equivalente - que denominamos **fatoração** do nó do tipo **OR**.

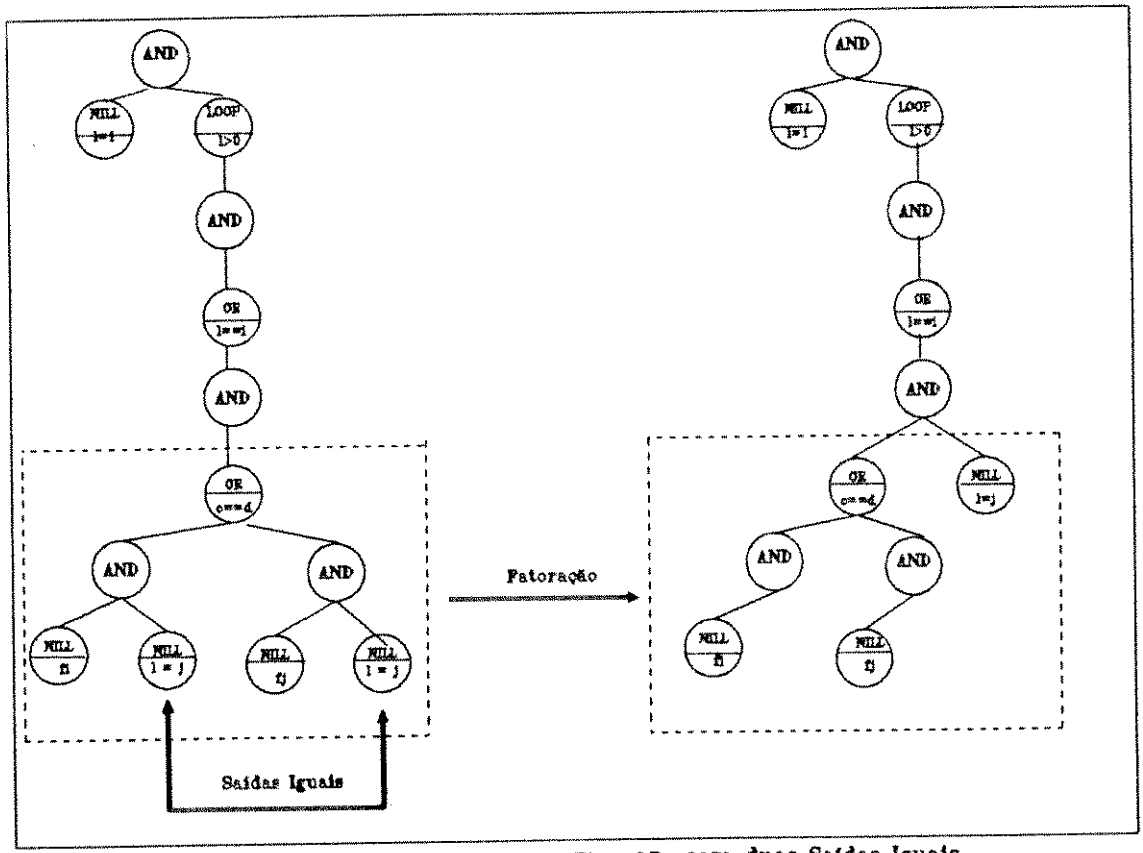

Figura 4.3 : Fatoração de um Nó do Tipo OR com duas Saídas Iguais

Em alguns casos, fatorar as **saídas** em nós do tipo **OR** implica inverter o predicado, isto é, negar o predicado associado ao nó. Deve-se lembrar que um nó do tipo **OR** da **PrimeTree** possui obrigatoriamente pelo menos um filho que é o da esquerda; o filho da esquerda corresponde ao lado "**then**" do "**if-then-else**" representado pelo nó, enquanto o da direita corresponde ao lado "**else**". O segundo filho de um nó do tipo **OR**, pode não existir. Assim, considerando todas as possibilidades de **fatoração** de nós do tipo **OR**, para eliminar um nó do tipo **NILL** que é uma saída de um dos filhos do nó do tipo **OR**, o seguinte algoritmo deve ser utilizado :

```
EliminaSaida (saida)
  . no_and = PT_Pai(saida)
  . se (PT_QtdFilho(no_and) > 1 ) :
        . PT_Remove(saida)
  . se não :
        . no_or = PT_Pai(no_and)
        . se PT_Tipo(no_or) é OR :
              . se PT_NFilho(no_and) é 2 :
                    . PT_Remove(saida)
                    . PT_Remove(no_and)
              . se não :
                    . no_2 = PT_Filho(no_or,2)
                    . se existe no_2 :
                          . predicativo =  negação do
                                           predicativo de no_or
                          . PT_DelRamo(no_2)
                    . se não :
                          . PT_DelRamo(no_or)
        . se não :
              . PT_Remove(saida)
```

Figura 4.4 : Algoritmo EliminaSaida

Para realizar a composição de **seqüências**, passando pela fatoração dos nós do tipo **OR**, utilizam-se os seguintes algoritmos :

```
. CompoeElimina (PrimeTree)
    . fatorou = 0, morto = 0
    . numseq = número de seqüências da PrimeTree
    . i = numseq
    . enquanto ( i > = 1 ) :
        . s_i = i-ésima seqüência da PrimeTree
        . a_i = atribuição "l=i"
        . no_j = PT_Localiza(PT_Raiz(PrimeTree), a_i)
        . se no_j não existe
            -> PT_DelRamo (s_i), morto = 1
        . caso contrário :
            . no_k = PT_Localiza(PT_Sucessor(no_j),a_i)
            . se no_k não existe
                -> a_i têm apenas 1 ocorrência
                . se  PT_Descendente(no_j,no_i) é falso
                    -> s_i não é recursiva
                    . no_comp = PT_Compoem(no_j,s_i)
                    . PT_Remove(s_i)
                    . no_fat= FatoraOrNode(PrimeTree,
                                no_fat,fatorou)
                . caso contrário :
                    . s_i é código morto :
                        -> PT_DelRamo(s_i), morto = 1
            . i = i - 1
    . numseq = número de seqüências da PrimeTree
    . se (numseq > 0) e (fatorou ou morto)
            -> CompoeElimina(PrimeTree)
```

Figura 4.5 : Algoritmo CompoeElimina

```
. FatoraOrNode(PrimeTree, ramo, fatorou)
    . Para cada filho do ramo :
        . se PT_tipo(filho)  é  do tipo OR
            . procura a saída da esquerda
                -> sai-esq = FatoraOrNode(PrimeTree,
                             PT_Filho(filho,1),fatorou)
            . se sai_esq existe :
                . se existe PT_Filho(filho,2)
                    .sai_dir = FatoraOrNode(PrimeTree,
                         PT_Filho(filho,2),fatorou)
                    . se existe sai_dir
                        . se sai_esq = sai_dir
                            . Insere novo nó NILL,
                                como irmão do nó OR
                            . Faz conteúdo do novo
                                nó  NILL ser igual
                                    à saída
                            . EliminaSaida(sai_dir)
                            . EliminaSaida(sai_esq)
                . saida = novo nó NILL
        . caso contrário :
            . se PT_Tipo(filho) == NILL e contém um
              assinalamento, então saida = atribuição à "1"
    . retorna(saida)
```

Figura 4.6 : Algoritmo FatoraOrNode

Os algoritmos **CompoeElimina** e **FatoraOrNode**, aplicados à **PrimeTree** da **Primeira Forma Normal** de um código, simplificam o número de ciclos, reduzem o uso do contador adicional e eliminam código morto. Deve-se notar que a eliminação de **código morto** e a **fatoração** de nós do tipo **OR** são operações que também reduzem o número de ocorrências de atribuições do tipo *"l=j"*; assim o critério de parada dos algoritmos inclui a não ocorrência de fatoração ou ausência de código morto, além da pesquisa completa de todas as atribuições do tipo *"l=j"* na **PrimeTree**.

O número de ocorrências de atribuições ao contador adicional *"l"* ainda pode ser reduzido sem que sejam realizadas composições. Se existirem **seqüências recursivas** na **PrimeTree**, estas podem ser transformadas em nós do tipo **LOOP**, de modo a reduzir o uso do contador adicional e melhorar a clareza do código estruturado final. Uma **seqüência** s_j é **recursiva** se em s_j existe pelo menos uma atribuição do tipo *"l=j"*, ou seja, s_j é uma das **seqüências** sucessoras de si mesma.

A Figura 4.7 ilustra uma **seqüência recursiva** e sua transformação em um nó do tipo **LOOP**. Na figura, a **seqüência** s_j contém apenas duas saídas, associadas a um único predicativo. A transformação é realizada substituindo-se o **corpo da seqüência** por um nó do tipo **LOOP** e um nó do tipo **NILL**, onde :

. nó do tipo **LOOP** : contém o predicativo associado ao nó do tipo **OR** em s_j cuja saída correspondente é *"l=j"*. Este nó possui um nó do tipo **AND** que é pai dos nós que formam o **corpo da seqüência** original.

. nó do tipo **NILL** : **saída** correspondente à atribuição *"l=i"*, *i diferente de j*.

Deve-se notar que a transformação realizada mantém a equivalência funcional e ainda reduz o número de ocorrências da atribuição *"l=j"*, o que poderá permitir novas composições.

Também nas substituições de **seqüências** recursivas por nós do tipo **LOOP** podem ocorrer inversões de predicativos, ou até mesmo a inclusão de um predicativo, caso a **seqüência** original não o possua. A Figura 4.8 ilustra um caso de **seqüência** recursiva sem predicativo associado. De fato, a **seqüência** da figura é um "**loop infinito**" [KERNIGHAN, 88], para reescrevê-la na forma de uma iteração associa-se um predicativo cujo resultado seja sempre verdadeiro (vide ilustração).

Consideramos neste trabalho que o corpo de uma **seqüência** s_j resulta em um nó do tipo **LOOP** com predicativo p_j se e somente se em s_j existe um único nó do tipo **NILL** correspondente a uma atribuição *"l=j"* e um único nó do tipo **NILL** correspondente a uma atribuição *"l=i"*, para qualquer ***i diferente de j***.

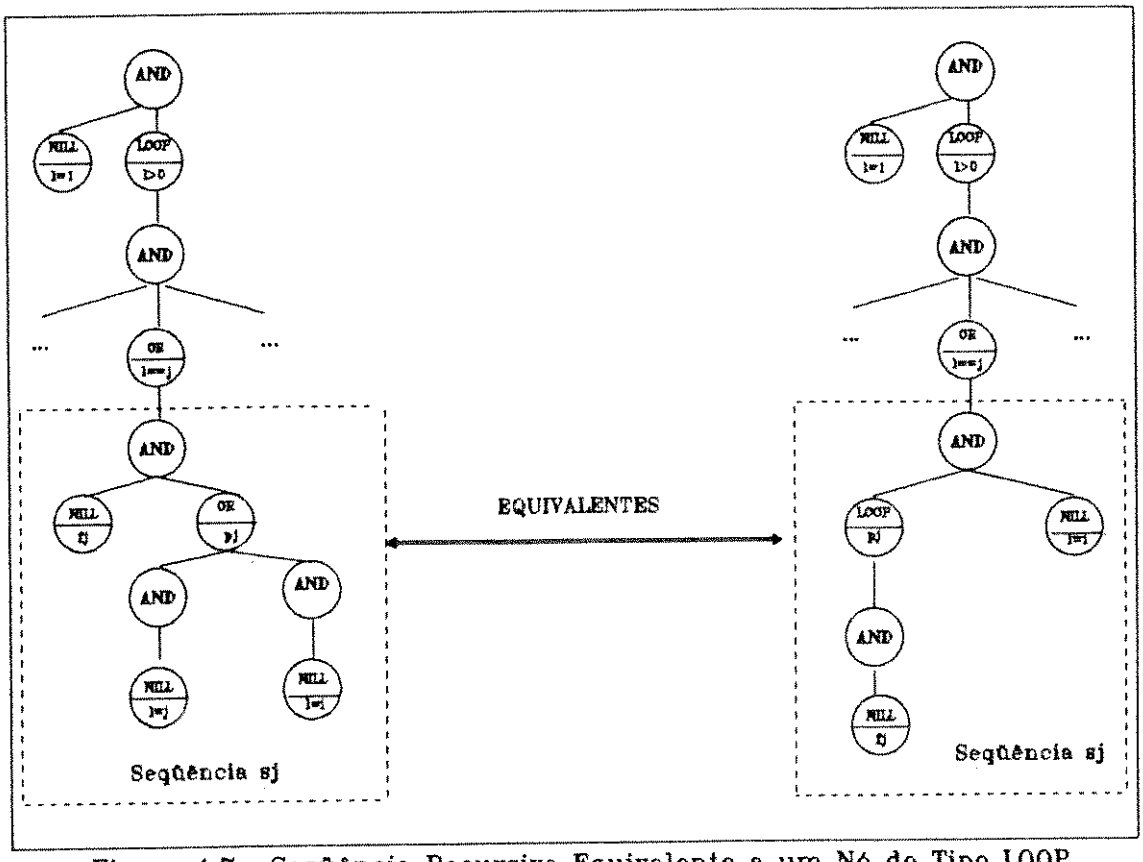

Figura 4.7 : Seqüência Recursiva Equivalente a um Nó do Tipo LOOP

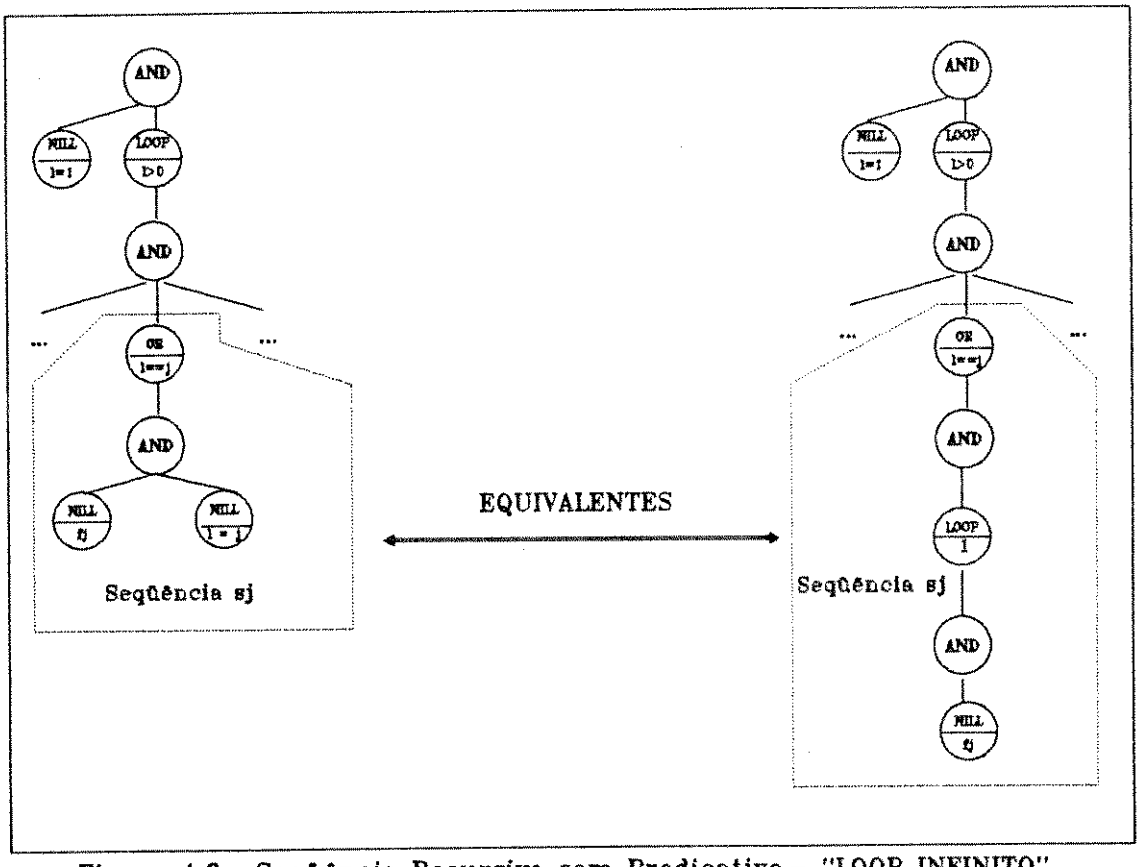

Figura 4.8 : Seqüência Recursiva sem Predicativo - "LOOP INFINITO"

Os algoritmos para encontrar e transformar **seqüências** recursivas em **seqüências** com iteração, são os seguintes :

```
. Identificação de Loops
    . Para cada seqüência s1 na PrimeTree :
        . temloop = IdLoops(PrimeTree,s1)
```

```
. IDLoops (p,s1)
    . se s1 contém apenas duas saídas :
        . se as duas saídas são diferentes
            . se apenas uma das saídas é "l=i", onde
                i é o número da seqüência s1
                TransformaLoop(p,s1,sai_igual,sai_dif)
                temloop = 1
    . caso contrário :
        . se existe apenas uma saída
            . se a saída é "l=i"
                 s1 é um loop infinito,
                          com predicativo "1"
                     transforma em loop
                     temloop = 1
        . caso contrário temloop = 0
    . retorna temloop
```

```
.TransformaLoop(p,s1,sai_igual,sai_dif)
    . predicativo = nó do tipo OR para o qual
                    PT_Descendente(no_OR,sai_igual)
    . se PT_Filho(no_OR,2) existe :
        . se PT_Descendente(no_OR,sai_igual)
            -> nega o predicativo
    . Cria novo nó do tipo LOOP, com o predicativo
    . Cria irmão NILL do novo LOOP com sai_dif
    . EliminaSaida(sai_igual)
    . EliminaSaida(sai_dif)
    . Incorpora o corpo da seqüência s1 ao novo LOOP
```

Figura 4.9 : Algoritmos para Tratamento de "LOOPs"

Deve-se observar que poderiam ter sido feitas outras considerações para identificar **seqüências** iterativas. Uma alternativa seria compor predicados [HOPCROFT, 79] para o caso de uma **seqüência** s_j com mais de uma saída do tipo *"l=j"* e apenas uma saída do tipo *"l=i"*, *i diferente de j*; ou, ainda, simplificar uma **seqüência** s_j com mais de uma saída *"l=i"*, *i diferente de j*, utilizando alguma transformação de estruturação sobre grafos, como o Método de Oulsnam [OULSNAM, 87].

No entanto, o objetivo do **Modelo de Estruturação** deste trabalho é fornecer uma base para ferramentas de reengenharia, especialmente para processos de partição de código [MACARIO, 92]. A redução de laços no programa por outros métodos poderia melhorar sua cognitividade, mas não levaria necessariamente a uma melhor identificação de partições do

código, pois o predicado associado a um laço é uma variável que causa interdependência entre os demais nós do mesmo laço [REZENDE, 92].

Aplicando os algoritmos de composição seguido pela identificação de "**loops**", segundo o **Modelo de Redução** da Figura 4.1 da seção anterior, até que não seja mais possível compor **seqüências** ou identificar novas iterações, obtém-se o código estruturado final que servirá de base a outras ferramentas de reengenharia.

4.2. Exemplo de Aplicação

Os **esquemas de decomposição** para as linguagens **COBOL**, **FORTRAN** e C obtêm o **delineamento de cada unidade de programa** do código fonte original com suas respectivas representações na **Primeira Forma Normal**.

O **Modelo de Redução** aplicado à representação da **Primeira Forma Normal** de cada **unidade de programa** permite obter uma nova representação também estruturada, porém mais clara e eficiente.

Muitas vezes no programa original podem estar embutidos laços que são delimitados por desvios em lugar de blocos de repetição. Ao aplicar o **esquema de decomposição** ao código original teremos um código seqüenciado que, embora estruturado, não representará nitidamente os blocos de repetição internos ao código. A aplicação do **Modelo de Redução** permitirá identificar este tipo de estrutura de controle no código.

Na Seção 3.1 foi descrito o **Esquema de Decomposição** para códigos fonte em linguagem **COBOL**. Nessa seção temos os resultados da normalização de cada **unidade de código** do programa dado como exemplo, que é o seguinte :

```
procedure division.
programa section.
par1.
        move 0 to p.
        move 1 to q.
        perform b.
        perform c.
        perform d.
        perform e.
        perform tudo.
        stop run.

tudo section.
a.
        move 0 to q.
        move 0 to p.
b.
        if p = 1
                go a.
        if q = 1
                go d.
c.
        add 1 to i.
d.
        if q = 1
                go a.
e.
        display "i = "i.
```

A **unidade de programa** "**d**", no exemplo acima o parágrafo "**d**", aparentemente é o controle de um laço cujo início deve ser a unidade "**a**", ou seja, o parágrafo "**a**".

Observando-se a **Primeira Forma Normal** da unidade "d", na Figura 4.10, a presença do laço é menos visível que no original. Aplicando-se o **Modelo de Redução** passo a passo pode-se observar como as estruturas são identificadas.

Por exemplo, ao final da quinta iteração do algoritmo "**CompoeElimina**", tem-se a **PrimeTree** da Figura 4.11. Nessa figura, nota-se que a **seqüência** s_{17} contém um nó do tipo OR com duas saídas iguais. Aplicando-se o algoritmo **FatoraOrNode** à **seqüência** s_{17} tem-se a **PrimeTree** da Figura 4.12.

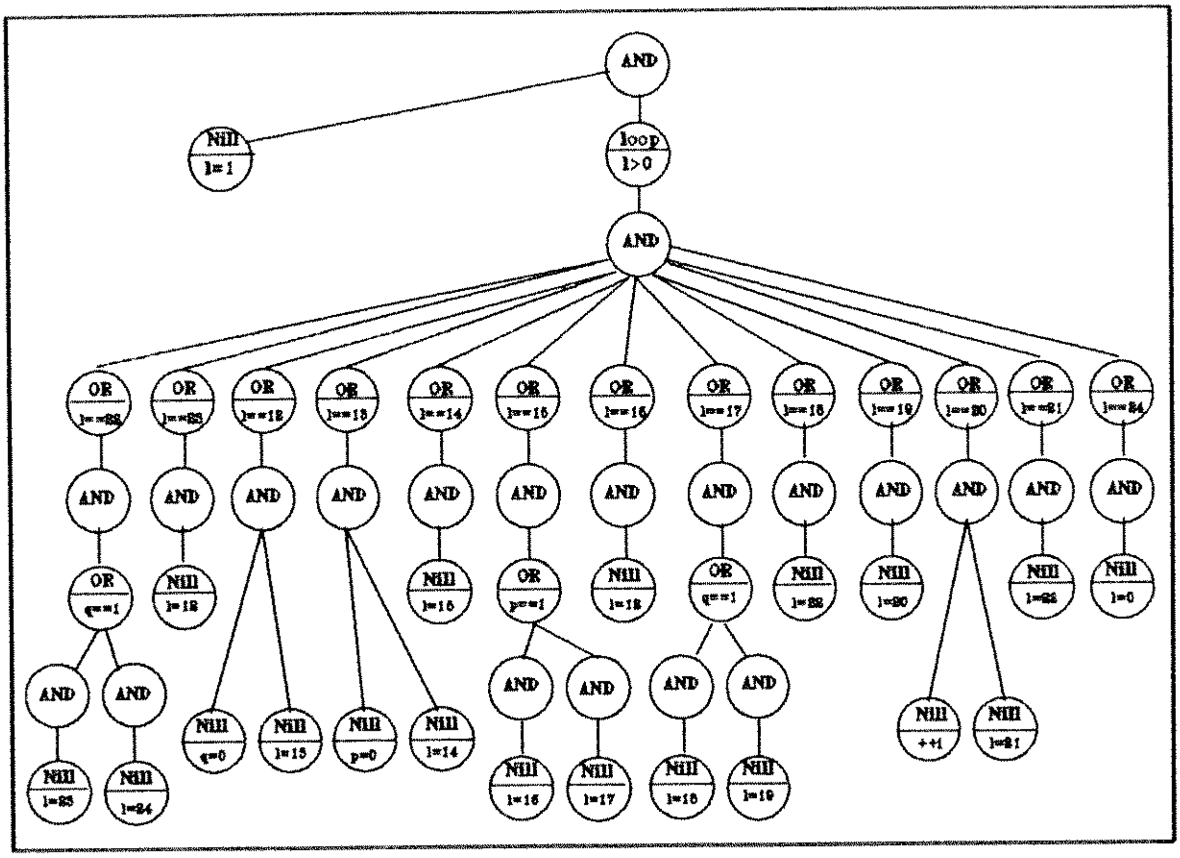

Figura 4.10 : Primeira Forma Normal da Unidade de Programa "d"

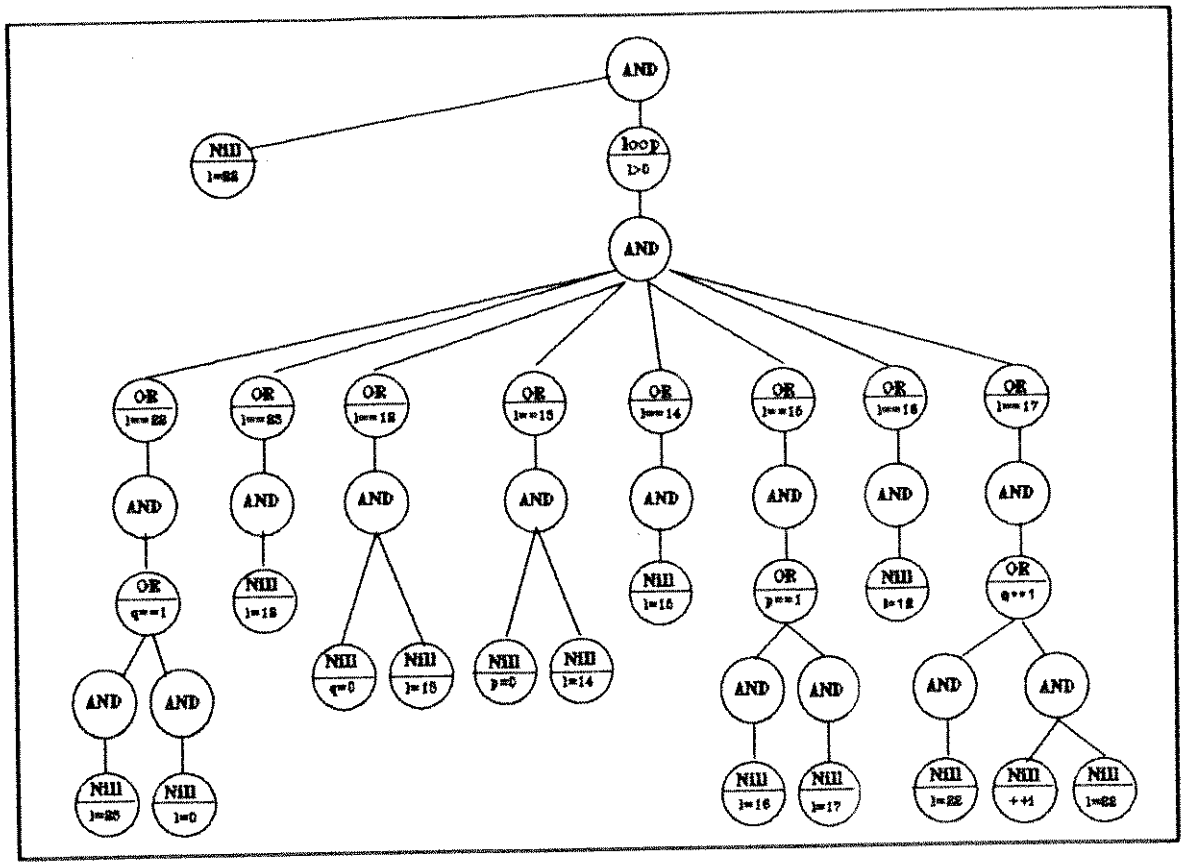

Figura 4.11 : Primetree "d" ao final da quinta iteração de "CompoeElimina"

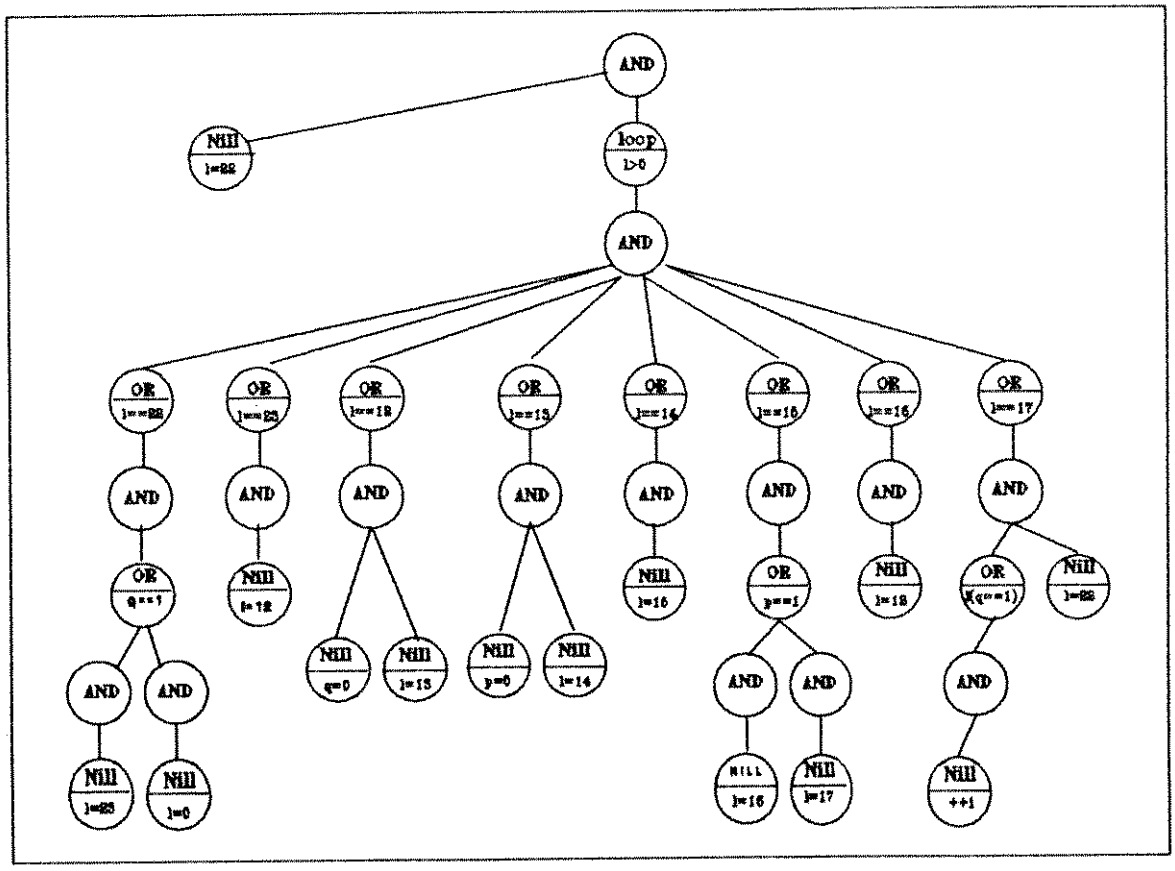

Figura 4.12 : Fatoração da Sequência s_{17} - PrimeTree "d"

Voltando a aplicar o algoritmo "**CompoeElimina**" ao final da décima quarta iteração teremos a **PrimeTree** da Figura 4.13. Deve-se notar que esta última **PrimeTree** possui duas atribuições do tipo *"1=22"* e duas atribuições do tipo *"1=12"*. A atribuição *"1=0"* jamais é substituída; ou seja, a **seqüência** ***so*** jamais é composta pois ***so*** não existe de fato (vide Seção 2.1.1). Logo, não é possível fazer novas composições.

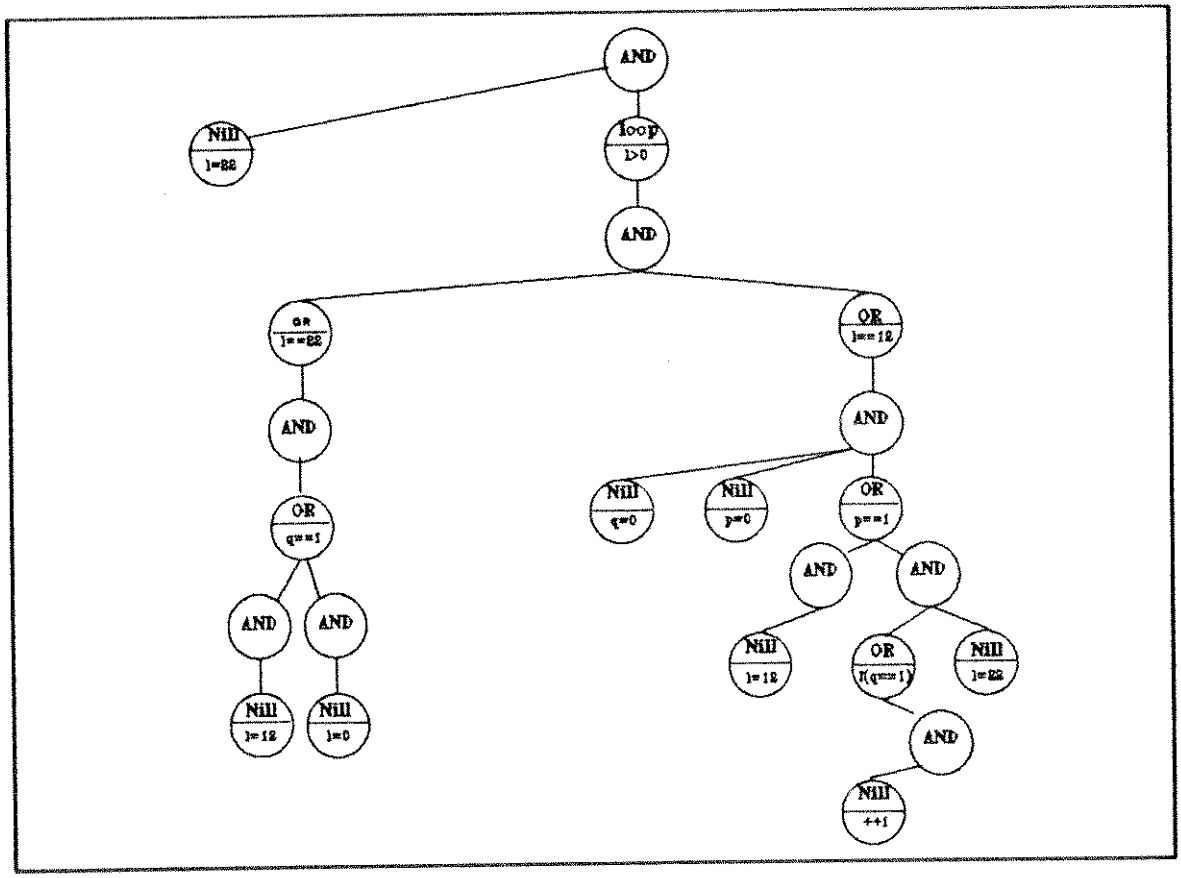

Figura 4.13 : PrimeTree "d" ao Final da Décima Quarta Iteração de "CompoeElimina"

No entanto, a **seqüência** ***s12*** é recursiva, contém uma única atribuição *"1=12"* e um predicativo *"p==1"* que controla a iteração embutida em ***s12***. Aplicando-se o algoritmo de identificação de "**loops**" tem-se a **PrimeTree** da Figura 4.14.

Novas composições podem ser aplicadas à **PrimeTree** da Figura 4.14 pois a **PrimeTree** contém uma única ocorrência da atribuição *"1=12"*; a composição resulta na **PrimeTree** da Figura 4.15.

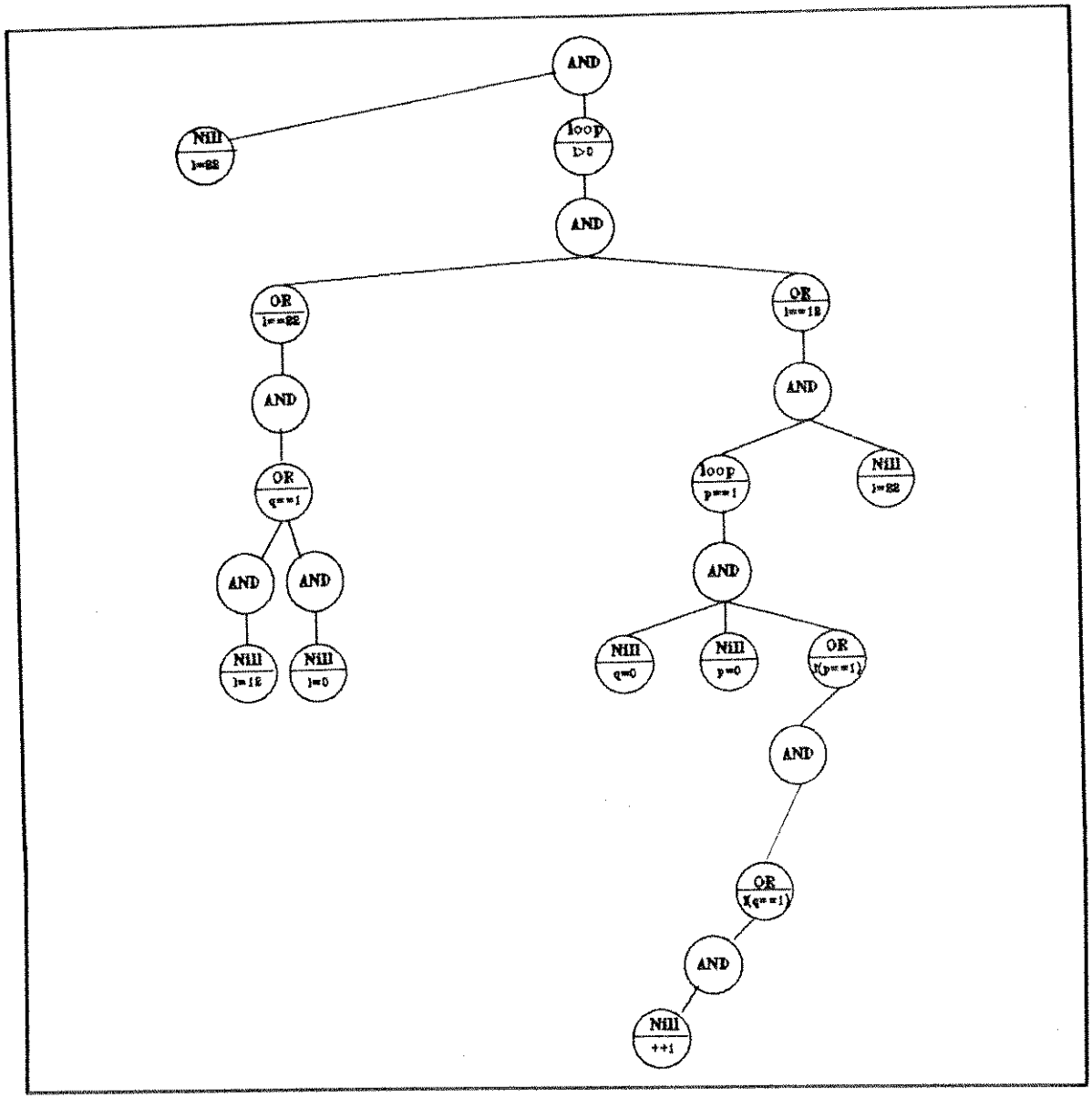

Figura 4.14 : Identifica "loop's" - PrimeTree "d"

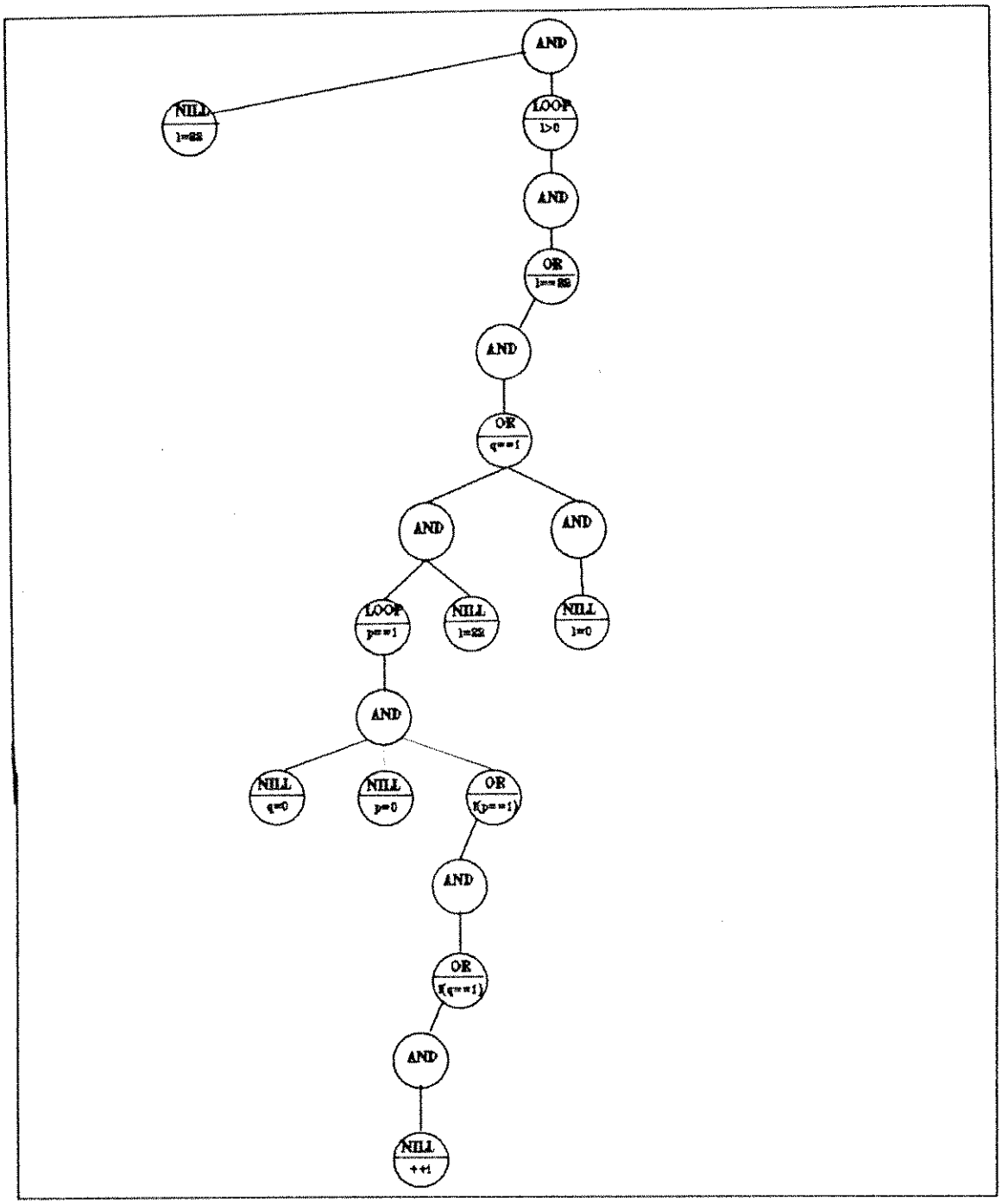

Figura 4.15 : Composição de s12 com s22 PrimeTree "d"

Na Figura 4.15 nota-se que a **seqüência** *s22* é recursiva e a iteração é controlada pelo predicativo *"q==1"*. Aplicando novamente os algoritmos de identificação de **"loops"** tem-se a **PrimeTree** da Figura 4.16, que contém uma única atribuição *"1=22"*. Realizando a última composição, para a **seqüência** *s22*, tem-se a **PrimeTree** da Figura 4.17.

A partir da Figura 4.17, pode-se escrever o código fonte do programa estruturado, da seguinte forma :

```
void d()
{
     do {
          if ( q == 1 ) {
               do {
                    p = 0;
                    q = 0;
                    if ( ! ( p == 1 ) ) {
                         if ( ! ( q == 1 ) ){
                              ++i;
                         }
                    }
               } while ( p == 1 );
          }
     } while ( q == 1 );
}
```

Comparando-se o código estruturado com o original, é possível notar que no original não existem iterações explícitas, não é possível reconhecer os predicados a elas associados, assim como não é possível reconhecer exatamente quais as linhas de código (funções) que fazem parte da **unidade de programa "d"**.

No entanto, o código resultante da aplicação do **Modelo de Estruturação** completo, após a normalização e a redução da estrutura do programa, permite ter uma idéia exata da estrutura do código.

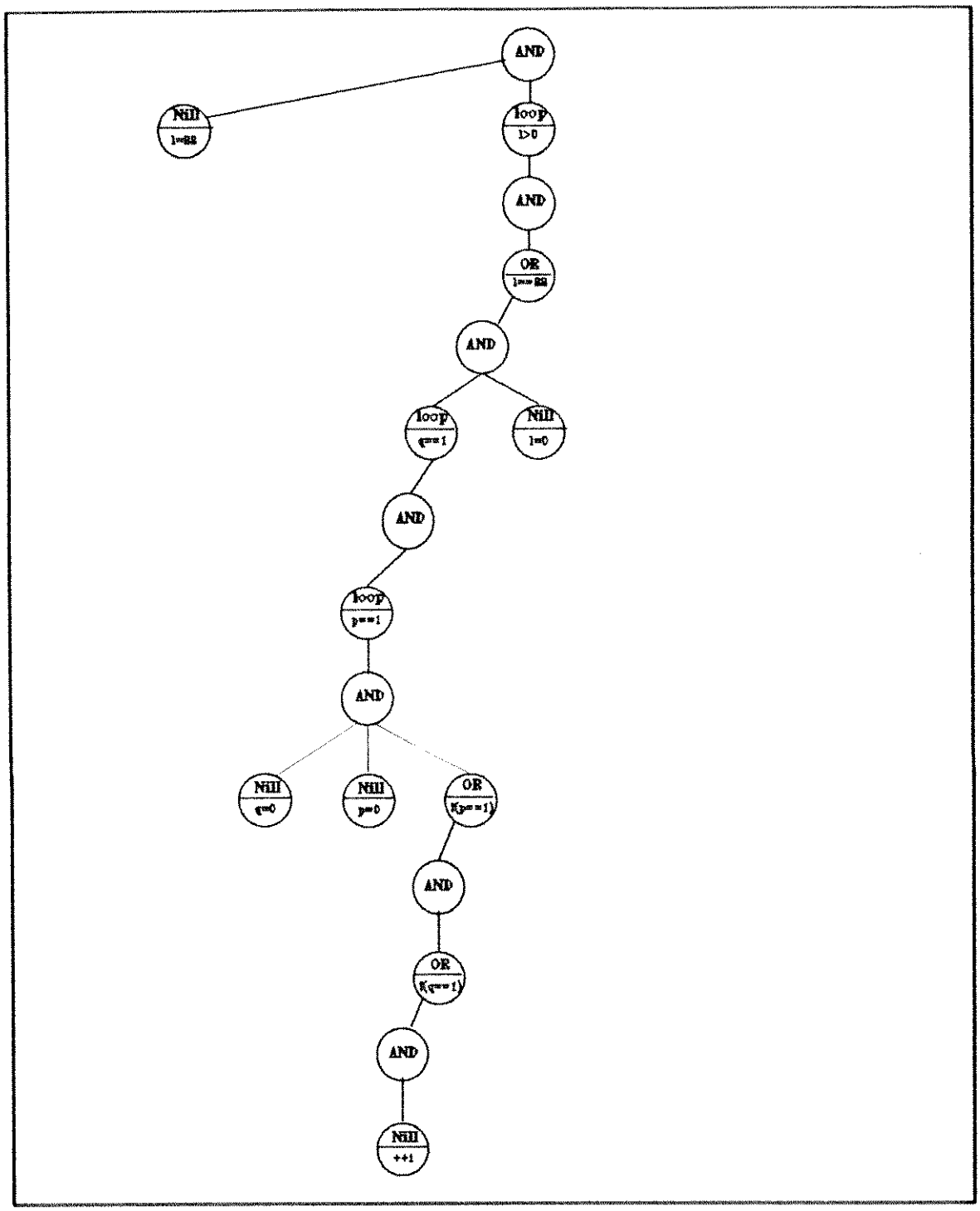

Figura 4.16 : Identifica "loop's" - sequência s_{18} PrimeTree "d"

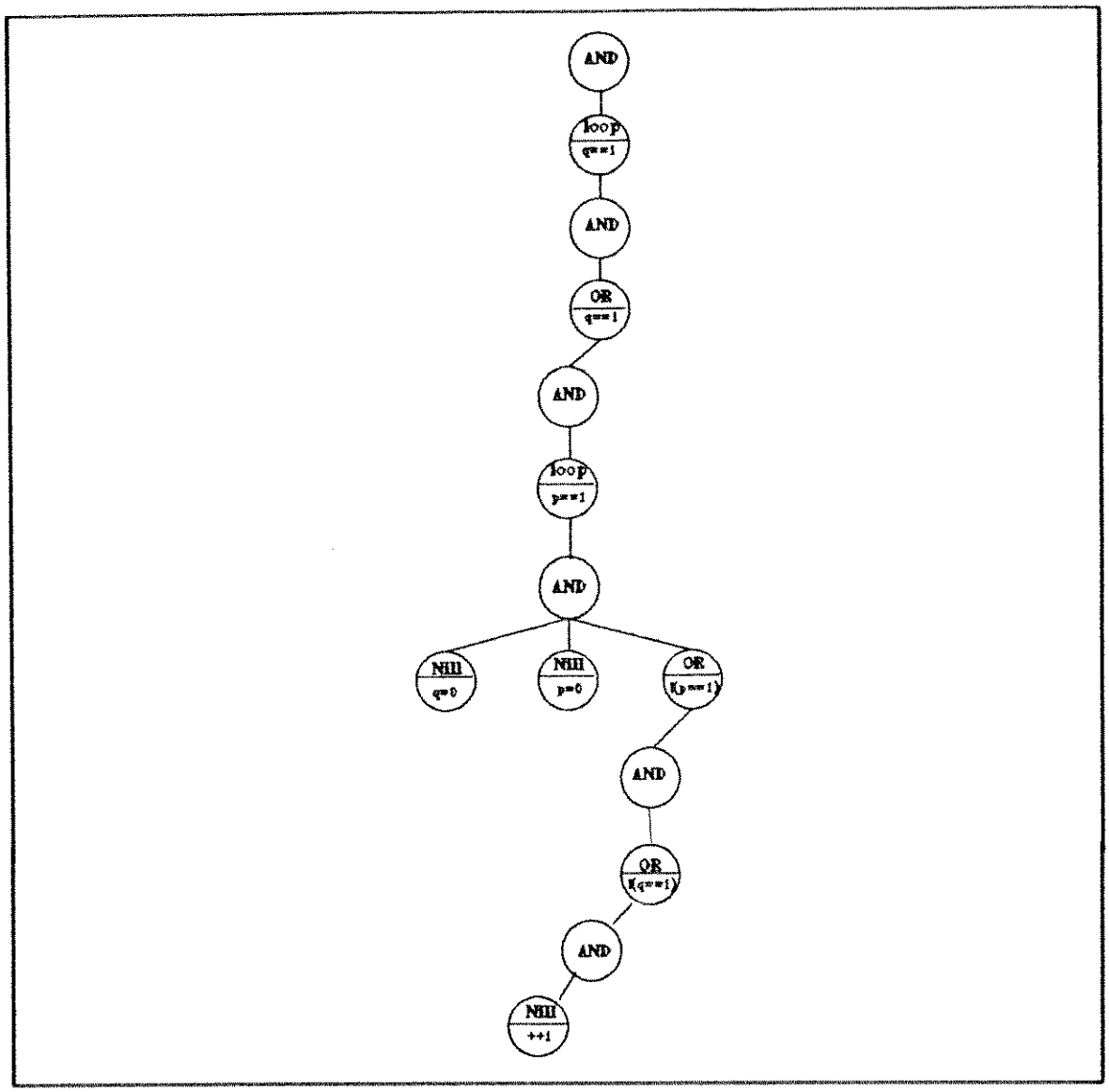

Figura 4.17 : PrimeTree "d" Reduzida - Forma Normal Reduzida

5. IMPLEMENTAÇAO DO MODELO DE ESTRUTURAÇAO E EXEMPLO DE APLICAÇAO

Neste capítulo são apresentadas algumas sugestões para implementação do **Modelo de Estruturação** sob ambiente **UNIX** e um exemplo de implementação (protótipos desenvolvidos); são apresentadas as ferramentas do ambiente **UNIX** necessárias à implementação, o plano de implementação e a arquitetura dos protótipos.

Como exemplo de aplicação, o código fonte de um produto de software é estruturado com uso dos protótipos implementados. Mostra-se como executar o processo de estruturação do código e são realizadas algumas análises sobre o código fonte original e o estruturado (complexidade estrutural e equivalência funcional entre os códigos original e estruturado).

5.1 Implementação do Modelo de Estruturação

O **Modelo de Estruturação** divide-se em duas etapas : obtenção da **Primeira Forma Normal** do código fonte, que corresponde ao **esquema de decomposição (seqüenciamento e delineamento de unidades)**; e obtenção da **Segunda Forma Normal** do código, que consiste na redução do uso da variável auxiliar, eliminação de código morto e identificação de blocos de iteração.

Automatizar a implementação de um **esquema de decomposição** implica obter um **analisador léxico** e **sintático** para a linguagem de programação do código fonte original e construir os **seqüenciamentos** para cada estrutura de controle, a partir desses analisadores.

Se a gramática da linguagem de programação do código fonte original é livre de contexto, ou assim puder ser representada, ela pode ser especificada por uma **BNF**, "Backus Normal Form" [HOPCROFT, 79].

Uma especificação **BNF** pode ser implementada por um gerador de analisador sintático (gerador de **"parser"**) do tipo **LR(k)** [AHO, 86] como o **YACC**, "Yet Another Compiler Compiler", que está disponível em ambientes **UNIX** [KERNIGHAN, 84] e gera um "parser" LALR ("Look Ahead Parsing", análise da esquerda para a direita e com uso do próximo símbolo terminal).

Especificando a gramática de uma linguagem com o uso do **YACC** incluem-se regras para descrever as estruturas sintáticas da linguagem, código para ser invocado quando uma regra é reconhecida e rotinas para realizar as entradas básicas [JOHNSON, 84-b] (por exemplo, a rotina de análise léxica).

O **YACC** gera uma função, em linguagem **C** portável, para controlar o processo de entrada, que é um analisador sintático (**"parser"**). O **"parser"** gerado controla as chamadas ao **analisador léxico**, que reconhece os símbolos terminais (**"tokens"**). Os **"tokens"** são organizados de acordo com as regras estruturais que compõem a

gramática. Quando uma das regras é reconhecida, os códigos relativos à redução da regra são invocados; esses códigos são denominados **ações**. As **ações** permitem retornar valores que podem ser utilizados por outras **ações** [JOHNSON, 84-b]; ações devem ser implementadas em **C**, assim como outras funções e operações.

O **analisador léxico** pode ser construído para cada aplicação ou ser gerado automaticamente. A geração automática, utilizando o "**lex**" do ambiente **UNIX** e, só pode ocorrer se a gramática da linguagem for composta apenas por expressões regulares [LESK, 84].

Utilizando o **YACC** e um **analisador léxico** compatível na implementação de um **esquema de decomposição**, a **ação** relativa ao reconhecimento de uma estrutura de controle corresponde ao **seqüenciamento** da estrutura, como na ilustração da Figura 5.1.

Essa forma de implementação facilita a obtenção dos algoritmos de **seqüenciamento** da primeira fase de um **esquema de decomposição**, exigindo poucas estruturas de dados para controle das transformações.

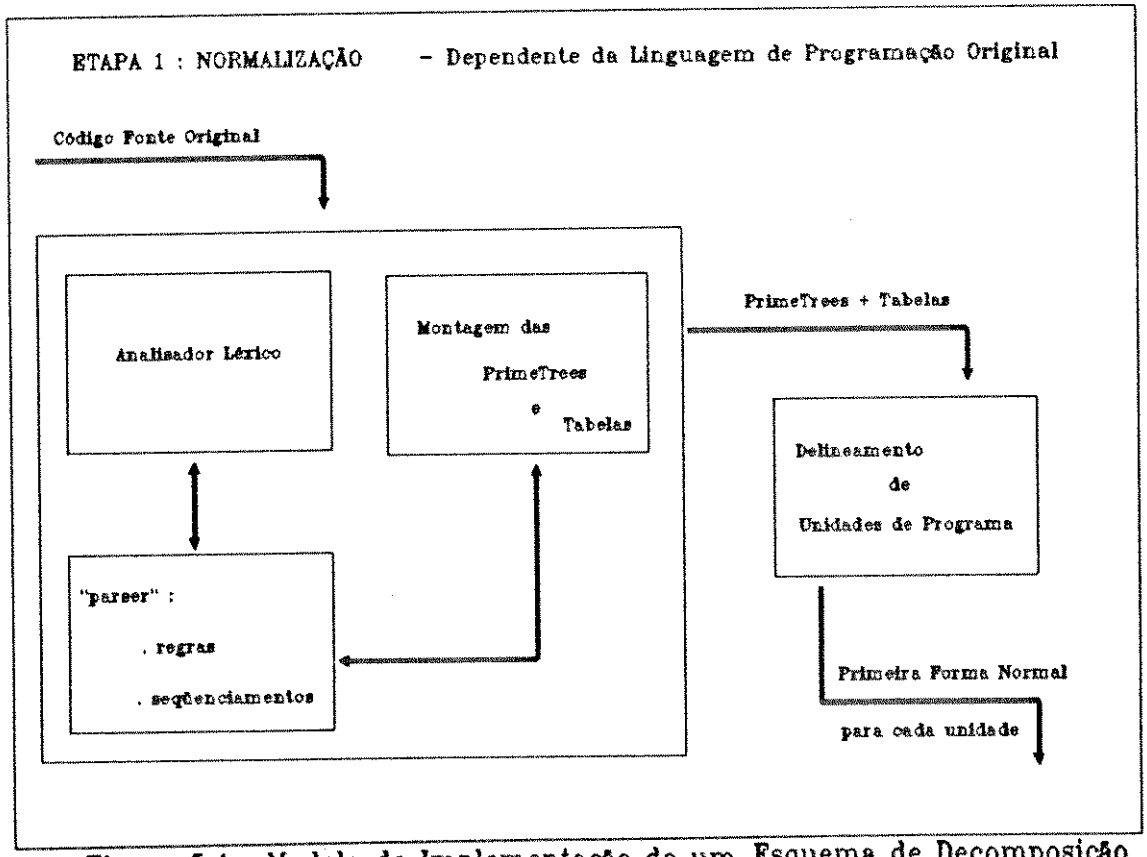

Figura 5.1 : Modelo de Implementação de um Esquema de Decomposição

Se o **esquema de decomposição** para a linguagem tratada possuir mais de uma fase, como para as linguagens **COBOL** e **FORTRAN**, as fases seguintes dependem apenas da(s) **PrimeTree**(s) e tabela(s) gerada(s) a partir das **ações** do **YACC**. Os algoritmos para as outras fases podem ser implementados sobre a(s) **PrimeTree**(s) e tabela(s) gerada(s), que pode(m) ser gravada(s) em arquivos utilizados para a comunicação com as outras fases, como ilustrado na Figura 5.1.

A implementação do **Modelo de Redução** independe completamente da linguagem de programação original. O modelo é diretamente implementável a partir dos algoritmos descritos na Seção 4.1, mediante o controle de entrada da(s) **PrimeTree**(s) que representa(m) a **Primeira Forma Normal** do código original, obtidas pela **Etapa 1**.

5.1.1. Protótipo Implementado

A Figura 5.2 ilustra o plano de implementação do **Modelo de Estruturação** [MOURA, 91], que prevê duas ferramentas para obtenção da **Primeira Forma Normal** de código - **C_FNC** e **COB_FNC**, e uma ferramenta para obtenção da **Forma Normal Reduzida** - **REDUCAO**.

A ferramenta **C_FNC** obtém a normalização de um código fonte em linguagem **C**, utilizando o **esquema de decomposição** descrito na Seção 3.3; a ferramenta **COB_FNC** obtém a normalização de um código fonte em linguagem **COBOL**, utilizando o **esquema de decomposição** descrito na Seção 3.1. A ferramenta **REDUCAO** corresponde à implementação do **Modelo de Redução** descrito na Seção 4.1.

As ferramentas **C_FNC** e **COB_FNC** produzem três arquivos, cujos nomes correspondem às **unidades de programa** do código fonte original :

. **<arquivo>.PTF** : contém uma representação da **PrimeTree** de uma **unidade de programa** do código fonte original em sua **Primeira Forma Normal**.

. **<arquivo>.HDR** : contém as declarações de dados, tipos e funções, que são necessárias ao código fonte da **unidade de programa** correspondente. Se o código fonte original estiver em uma linguagem de programação diferente de **C** este arquivo corresponde às traduções de estruturas de dados e declarações de tipos de funções, como para a ferramenta **COB_FNC**.

. **<arquivo>.FNC** : contém um programa em linguagem **C** que corresponde à representação da **Primeira Forma Normal** da **unidade de programa**. O cabeçalho deste arquivo apresenta a data da normalização, nome da função e a estimativa do número de linhas de código do código fonte padronizado.

A comunicação entre as etapas de normalização e redução é realizada através dos arquivos com extensão "**.PTF**" e "**.HDR**". O arquivo com extensão "**.FNC**" destina-se ao teste da implementação e informação sobre o número de linhas de código do código padronizado.

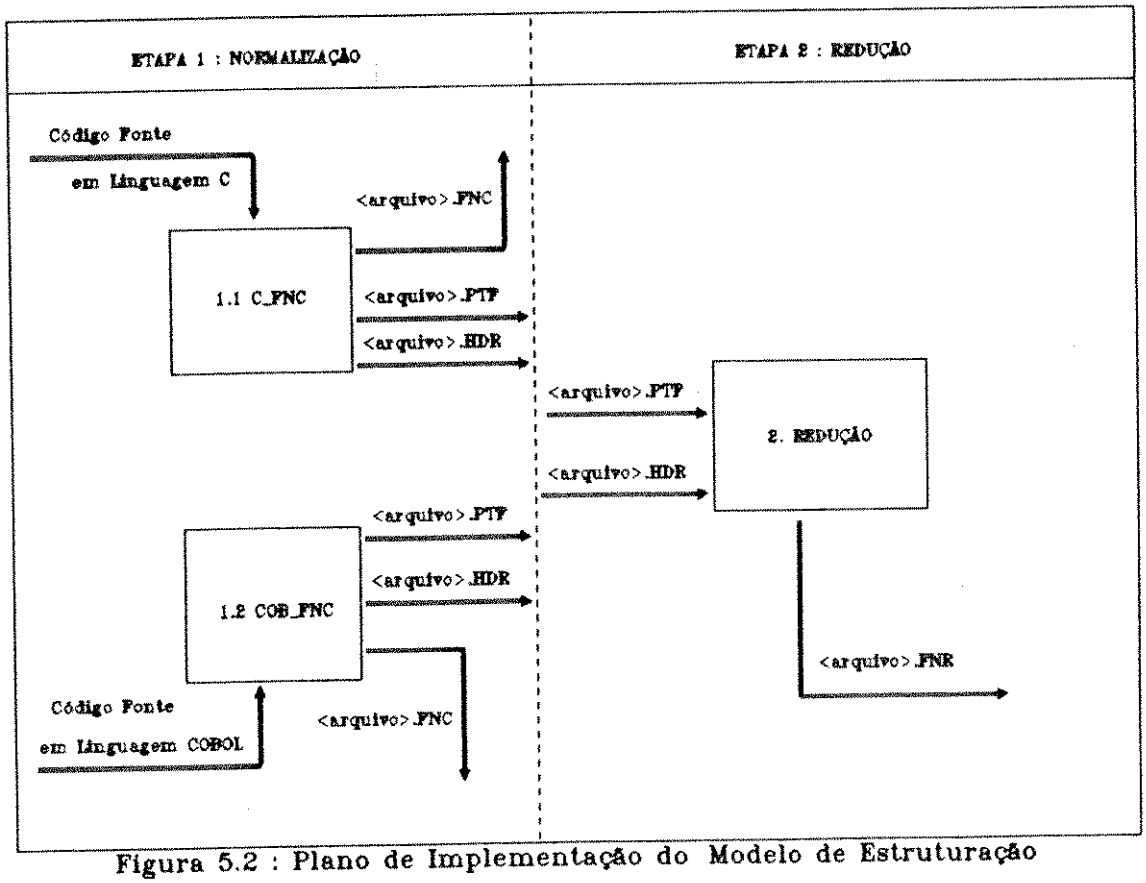

Figura 5.2 : Plano de Implementação do Modelo de Estruturação

A ferramenta **REDUCAO** trabalha sobre a **PrimeTree** armazenada no arquivo com extensão ".PTF" e gera uma nova **PrimeTree**. A nova **PrimeTree** e o arquivo com extensão ".HDR" dão origem ao seguinte arquivo :

. **<arquivo>.FNR** : contém um programa em linguagem **C** que corresponde à **Forma Normal Reduzida** da unidade de programa.

Deste plano de implementação foram desenvolvidos, implementados e testados os protótipos das ferramentas **C_FNC** e **REDUCAO**. Para a ferramenta **COB_FNC** foi implementado um núcleo mínimo, sob a especificação de um **COBOL ANSI**, com o objetivo de testar o **esquema de decomposição** proposto na Seção 3.1; o protótipo conta com algumas traduções de estruturas de dados e funções básicas [CHAIM, 92].

A ferramenta **COB_FNC** é um tradutor fonte a fonte, para o qual está sendo desenvolvida uma biblioteca de operadores; os operadores mapeiam funções embutidas em comandos da linguagem **COBOL**, traduções e inicialização de dados [CHAIM, 92]. Na Figura 5.3 pode-se observar um esquema simplificado do que seria a arquitetura da ferramenta **COB_FNC**. A parte relativa ao **esquema de decomposição** refere-se à "**montagem das PrimeTrees e Tabela Ro**" e ao "**delineamento de unidades**".

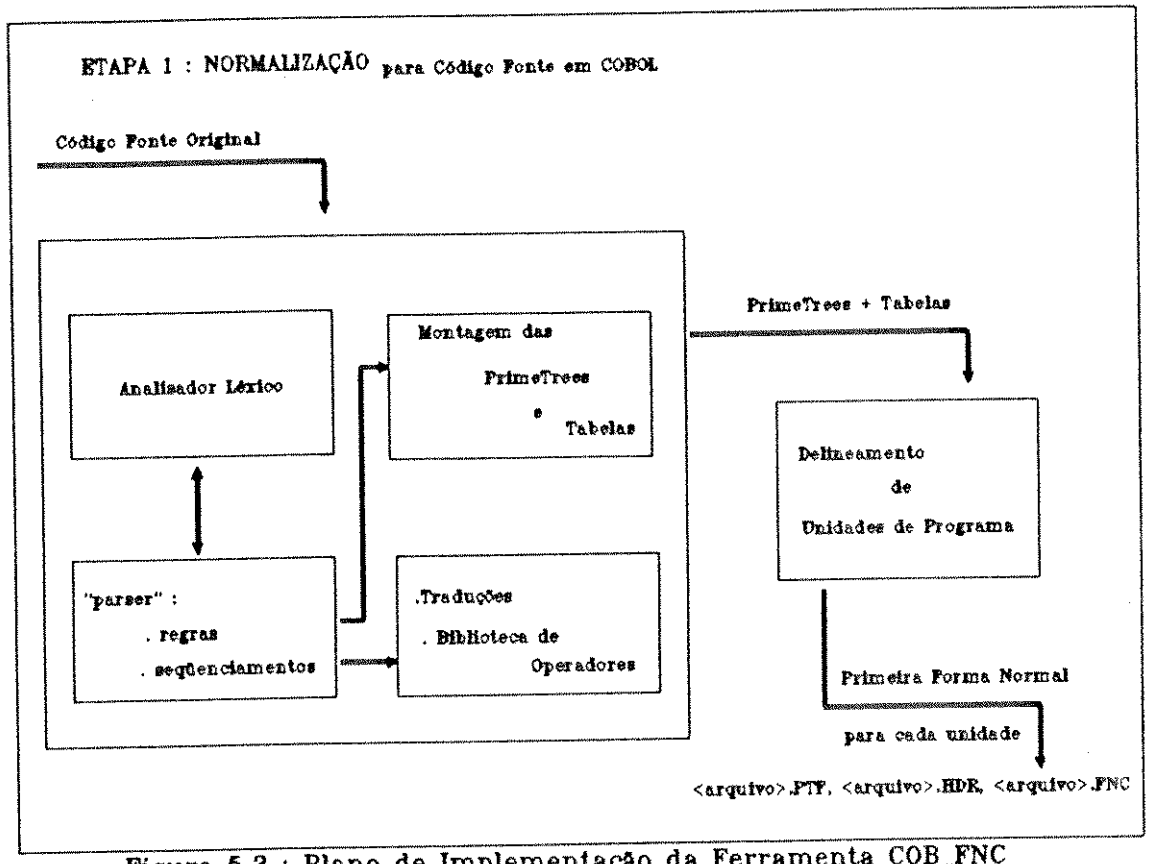

Figura 5.3 : Plano de Implementação da Ferramenta COB_FNC

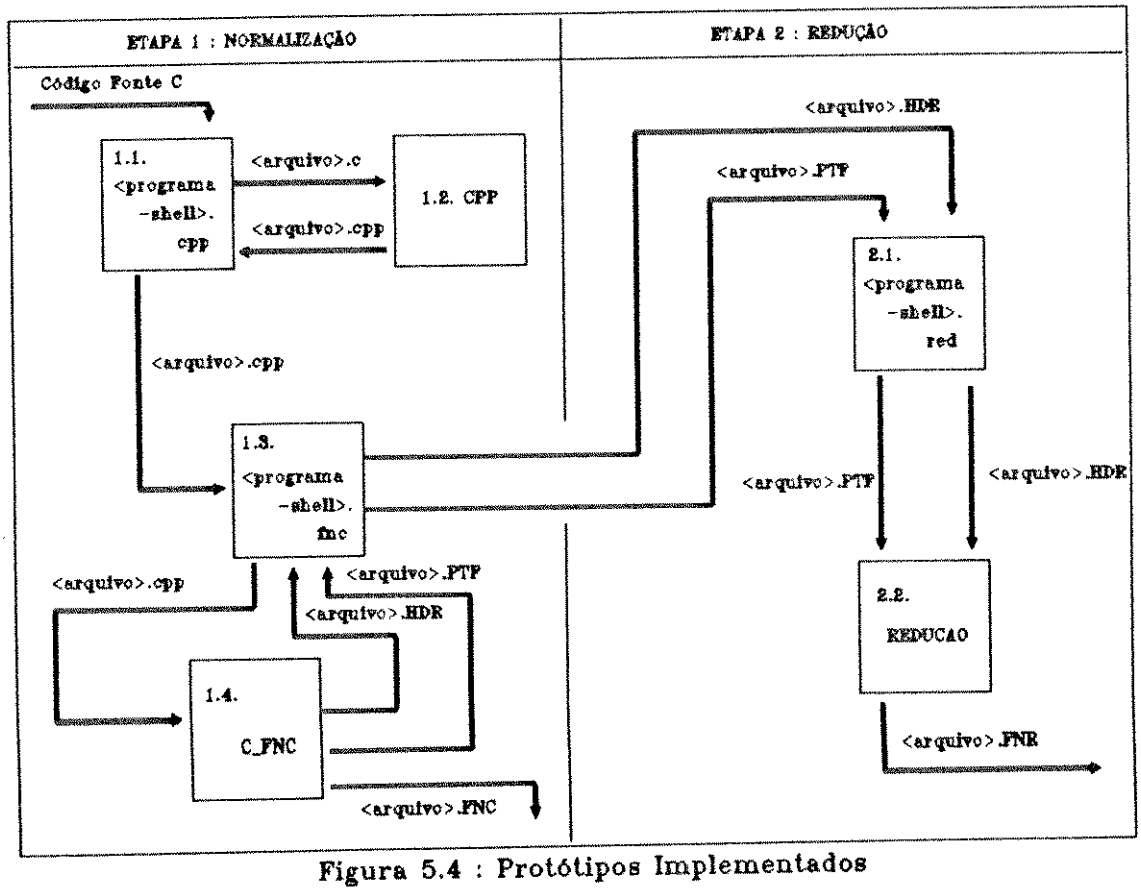

Figura 5.4 : Protótipos Implementados

A Figura 5.4 ilustra os protótipos das ferramentas **C_FNC** e **REDUCAO** correspondentes, respectivamente, às duas etapas do **Modelo de Estruturação** : normalização e redução.

Pode-se observar na Figura 5.4 que apenas os modelos foram implementados, pois os protótipos não possuem uma interface adequada. Para utilizá-los a interface para a aplicação deve ser construída através de programas **"shell"**. Denominam-se programas **"shell"** aos "scripts" construídos comando a comando que são interpretados e executados um a um pelo **"shell"** do ambiente **UNIX** [KERNIGHAN, 84].

Logo, para estruturar o código fonte de uma aplicação em linguagem **C**, que pode estar dividida em vários arquivos, deverá ser construído um programa **"shell"** para gerenciar cada passo do processo.

O protótipo **C_FNC** analisa um único arquivo fonte pré-processado, que pode conter qualquer número de **unidades de programa**. O pré-processamento do arquivo é necessário para que todas as declarações de tipos e dados estejam disponíveis para a análise do código; pois, alguns **seqüenciamentos** exigem a criação de novas variáveis, e seus tipos devem ser conhecidos (vide **seqüenciamentos** no **Apêndice C**).

Logo, uma execução de **C_FNC** deve ser precedida de pré-processamento e gerará tantos arquivos quanto for o número de unidades do programa original.

O protótipo **REDUCAO** analisa um único arquivo : arquivo com extensão **".PTF"**, que contém uma única **PrimeTree**. A execução gera um único arquivo (com extensão **".FNR"**) fazendo uso da **PrimeTree** reduzida no processo e das informações do arquivo de declarações (arquivo com extensão **".HDR"**).

A seguir são descritas as arquiteturas dos protótipos implementados e as evoluções necessárias.

Arquitetura de C_FNC

O protótipo **C_FNC** obtém a **Primeira Forma Normal** de cada unidade de programa de um código fonte em linguagem C.

A base da arquitetura do protótipo consiste nos analisadores léxico e sintático e em rotinas de manutenção da tabela de símbolos e pilhas de controle. Esta base foi adaptada a partir do código fonte do compilador C portável, PCC - **"Portable C Compiler"**, de domínio público, descrito por Johnson [JOHNSON, 84-a].

O **PCC** é dividido em dois passos, sendo aproximadamente 75 % de seu código independente de máquina [JOHNSON, 84-a]. O primeiro passo do **PCC** é responsável pela análise léxica e sintática, manutenção da tabela de símbolos, construção de "parse-trees" para expressões e controle de desvios [JOHNSON, 84-a].

Os arquivos fonte do **PCC** correspondentes ao primeiro passo foram adaptados para o protótipo **C_FNC**; o segundo passo não foi utilizado, dado que é o responsável pela geração de código executável.

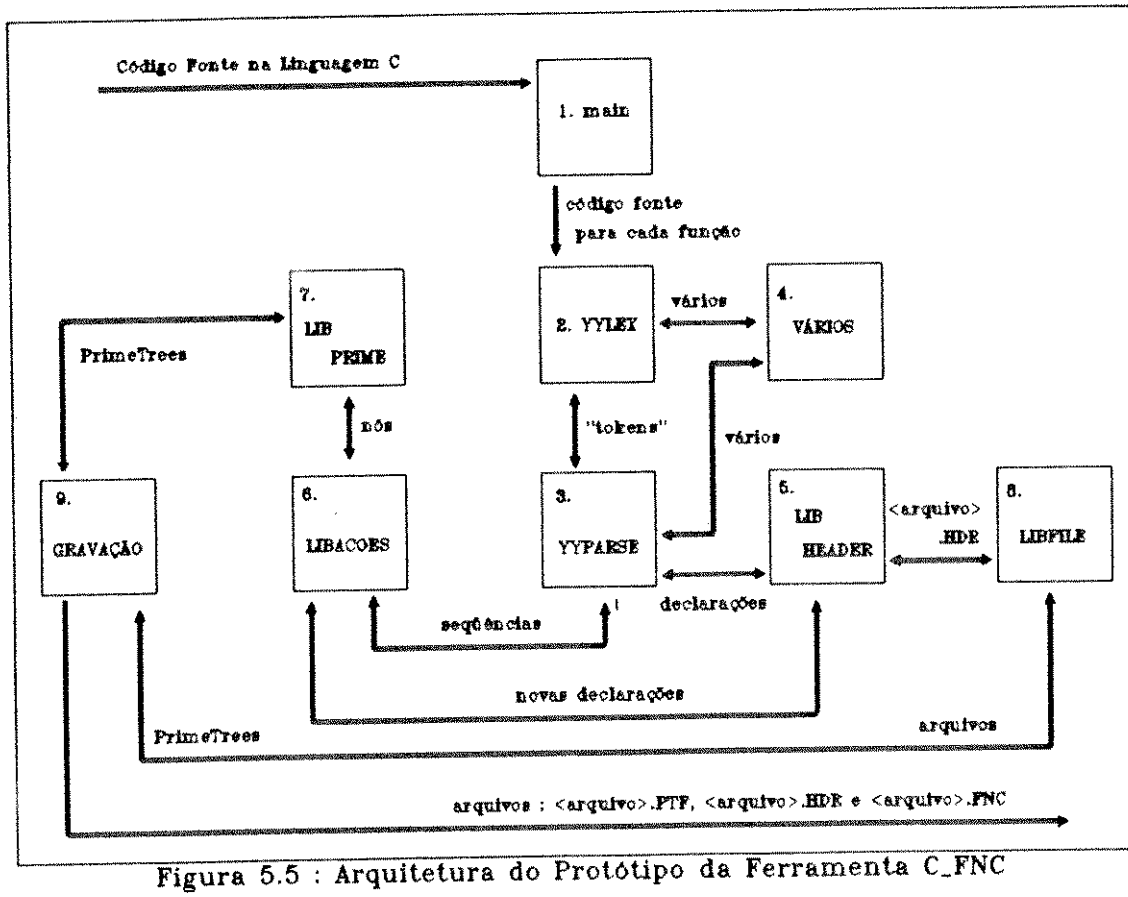

Figura 5.5 : Arquitetura do Protótipo da Ferramenta C_FNC

A Figura 5.5 ilustra a arquitetura do protótipo **C_FNC** esquematizado por nove módulos básicos, que são :

1. **"main"** : responsável pelo controle de rotinas de inicialização (por exemplo, criação de variáveis, alocações de memória, etc), primeira chamada do **analisador sintático** e rotinas de finalização (por exemplo, a gravação da última **PrimeTree** obtida).

2. **"yylex" : analisador léxico**. Neste analisador foram mantidos alguns controles do PCC, devido ao **"parser"** gerado ser "bottom-up" e a semântica da linguagem C ser "top-down" [JOHNSON, 84-a]. Alguns símbolos, como "typedefs", reconhecidos neste analisador, são encaminhados para as rotinas de controle do arquivo de declarações (arquivo com extensão ".HDR").

3. **"yyparse"** : analisador sintático gerado pelo **YACC**. As **ações** para cada regra reconhecida controlam os **seqüenciamentos** para a obtenção da **Primeira Forma Normal**.

4. **"vários"** : algumas rotinas de controle, que foram mantidas do PCC original, como gerenciamento da tabela de símbolos e a montagem das "parse-trees" de expressões.

5. **"libheader"** : biblioteca de funções que controlam a montagem do arquivo de declarações (**<arquivo>**.HDR).

6. **"libacoes"** : biblioteca dos **seqüenciamentos** das estruturas de controle para a linguagem C; são obtidos de acordo com as sugestões do **Apêndice C**.

7. "**libprime**" : biblioteca contendo os operadores do tipo abstrato **PrimeTree**, relacionados na Seção 2.1.5.

8. "**libfile**" : biblioteca de funções para ler e gravar os arquivos manipulados pelo processo de estruturação.

9. "**gravacao**" : rotina de finalização; encarrega-se de controlar a gravação dos últimos resultados obtidos na normalização.

Desta implementação são reusáveis as bibliotecas "**libprime**" e "**libfile**", pelo protótipo **REDUCAO** e ferramentas que obtenham a **Primeira Forma Normal** para códigos fonte em outras linguagens; por exemplo, a ferramenta **COB_FNC**. Os demais módulos são completamente dependentes da linguagem de programação do código fonte original, neste caso a linguagem **C**.

O protótipo é completamente portável e suporta códigos fonte em linguagem **C** compatíveis com as especificações da linguagem para o **PCC** [JOHNSON, 84-a]. O código fonte na versão atual conta com 22031 linhas de código, das quais 9625 correspondem ao reuso do passo 1 do PCC. O código executável é de 309K, em um equipamento MX-850 (ELEBRA) - 32 bits, 6 MegaBytes de RAM, 0.75 MIP, e sistema operacional ULTRIX´32 (UNIX).

Arquitetura de REDUCAO

O protótipo **REDUCAO** é a implementação dos algoritmos do **Modelo de Redução** (descritos na Seção 4.1.) em linguagem **C** e sob ambiente **UNIX**.

A Figura 5.6 ilustra a arquitetura do protótipo, que divide-se em nove módulos :

1. "**main**" : controla a leitura e gravação de arquivos e, também controla as chamadas à composição de programas primos ("**CompoeElimina**") e à identificação de nós do tipo **LOOP** ("**IdentLoops**"). Este módulo decide quando o processo de redução é encerrado, realiza as chamadas a "**CompoeElimina**" e a seguir a "**IdentLoops**", voltando a repeti-las se e somente se algum nó do tipo **LOOP** é encontrado e transformado (vide **Modelo de Redução**, Seção 4.1).

2. "**CompoeElimina**" : é a implementação do algoritmo do mesmo nome descrito na Seção 4.1, que é o responsável pelas composições de programas primos.

3. "**FatoraOrNode**" : simplifica o número de atribuições do tipo "**l=i**"; corresponde à implementação do algoritmo de mesmo nome descrito na Seção 4.1.

4. "**IdentLoops**" : identifica seqüências em uma **PrimeTree** reduzida, que são passadas ao módulo "**IDLoops**".

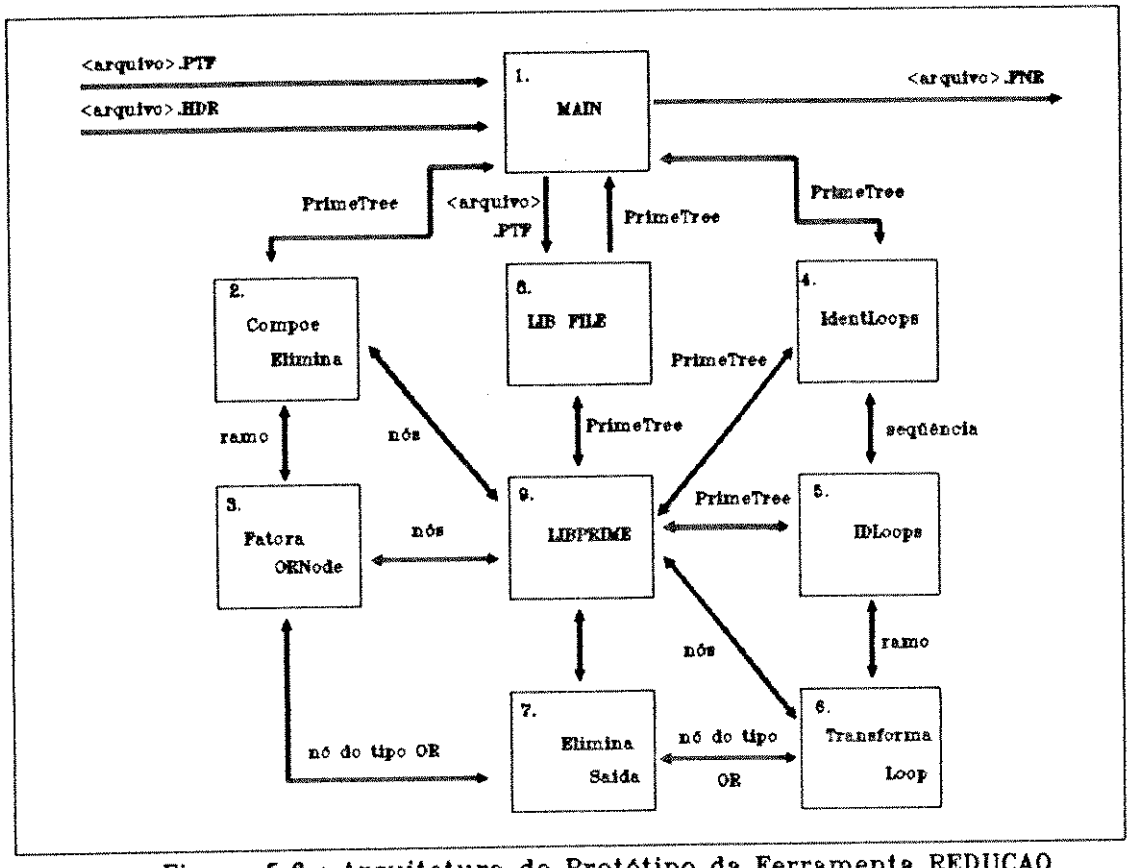

Figura 5.6 : Arquitetura do Protótipo da Ferramenta REDUCAO

5. "IDLoops" : verifica se uma **seqüência** corresponde a um nó do tipo **LOOP**; conforme definição e algoritmo da Seção 4.1.

6. "**TransformaLoop**" : a partir de um ramo da **PrimeTree** constrói um nó do tipo **LOOP**; o algoritmo implementado corresponde ao descrito na Seção 4.1.

7. "**EliminaSaida**" : inverte o predicativo de um nó do tipo **OR**, se necessário; corresponde à implementação do algoritmo de mesmo nome descrito na Seção 4.1.

8. "**libfile**" : biblioteca de funções que manipulam os arquivos tratados pelo protótipo.

9. "**libprime**" : biblioteca dos operadores do tipo abstrato **PrimeTree**; relacionados na Seção 2.1.5.

O protótipo é completamente portável; em sua versão atual conta com 2927 linhas de código e gera um executável de 38K, em um equipamento MX-850 (ELEBRA) - 32 bits, 6 MegaBytes de RAM, 0.75 MIPS e sistema operacional ULTRIX´32 (UNIX).

Evoluções Necessárias

Os protótipos implementados possuem restrições que dificultam o seu uso.

A seguir, são apresentadas as principais restrições presentes na versão atual dos protótipos que levam à necessidade de evolução :

. **Interface**

Os protótipos implementados não possuem um gerenciador de interface. Os programas "shell" utilizados para controlar as execuções, e, conseqüentemente, gerenciar os arquivos intermediários criados, devem ser substituídos por um processo automático.

A gerência da execução poderia ser feita pela interpretação automática das "makefiles" [KERNIGHAN, 84]; ou, a relação de arquivos e diretórios poderia ser fornecida pelo usuário através de um gerenciador de interface auto-explicativo.

O gerenciador de interface controlaria o processo de comunicação entre as ferramentas de pré-processamento, **C_FNC** e **REDUCAO**, fornecendo os resultados intermediários mais importantes : memória necessária para executar os processos, número de **seqüências** do código fonte original, presença de código morto, erros em inclusão de arquivos para pré-processamento, etc.

. **Construção de um Pré-processador**

Existe uma forte dependência do protótipo **C_FNC** em relação à ferramenta **"cpp"** (pré-processador do ambiente UNIX).

A ferramenta **"cpp"** não fornece apenas o resultado de pré-processamento necessário ao protótipo, mas inclui várias informações dispensáveis; por exemplo, substitui toda referência a entrada e saída padrão por ponteiros para os arquivos controlados pelo sistema. Logo, é necessário construir uma ferramenta que se encarregue do pré-processamento, ou incluí-lo no protótipo.

. **Gerenciamento de memória**

Os protótipos trabalham com as **PrimeTrees** em memória. Logo, o uso dos protótipos pode se tornar inviável se o código fonte analisado for excessivamente grande e o equipamento não dispuser de memória suficiente, ou for necessário realizar muitos "swappings".

Pode-se estimar o espaço de memória necessário a partir de uma pré-decomposição do código fonte não gerando qualquer alocação de memória. Esta estimativa permitiria verificar se existe disponibilidade de memória para o processo; no entanto, apenas essa informação não seria

suficiente.

O problema poderia ser resolvido mediante a reimplementação do gerenciamento de memória em conjunto com as estimativas, usando partições das **PrimeTrees** e outras alocações necessárias, em disco.

. **"Porting" e evolução de C_FNC**

Embora o protótipo implementado seja completamente portável, ele não pode ser aplicado a todo código fonte em linguagem **C**.

O protótipo gera o código estruturado funcionalmente equivalente ao código original em linguagem C se e somente se o código original é compatível com o compilador portável **PCC**.

Por exemplo, a seguinte expressão :

```
y = x + a + f(&x) +d;
```

Para compiladores compatíveis com o PCC, a chamada à função "f" acima tem precedência de execução em relação ao restante da expressão. A decomposição (**seqüenciamento**) desta expressão de acordo com o protótipo **C_FNC** é :

```
f_x = f(&x);
y = x + a + f_x + d;
```

No entanto, em outros compiladores C as precedências em expressões podem ser outras; por exemplo, a mesma expressão poderia ter a seguinte decomposição :

```
p1 = x      + a;
p2 = f(&x)  + d;
y  = p1     + p2;
```

Se a precedência acima fosse a correta, então a decomposição anterior não manteria a equivalência funcional com a expressão original pois o valor da variável "x" seria alterado na segunda parte da expressão.

Logo, para que o protótipo **C_FNC** possa ser aplicado a qualquer código **C** é necessário distinguir a regra de formação de expressões em cada compilador e implementar os **seqüenciamentos** de acordo com a especificação do compilador original.

5.2. Exemplo de Aplicação dos Protótipos

O produto de software utilizado para a aplicação dos protótipos é o módulo **SPLINE** do Software NTIA [NTIA, 92] cujo código fonte é implementado em linguagem **C** sob ambiente **UNIX**.

O módulo **SPLINE** calcula pontos de interpolação para um conjunto de coordenadas (x,y), pelo método "spline cúbica". Ele é composto por quinze funções (**unidades de programa**), distribuídas em seis arquivos fonte, da seguinte forma :

. arquivo **"getpar.c"** : contém as funções responsáveis pela interpretação dos parâmetros para execução do programa. E composto por três funções : "getpar", "number" e "usage".

. arquivo **"lerobs.c"** : contém a função "lerobs" responsável pela recuperação de dados.

. arquivo **"outhdr.c"** : contém a função "outhdr", que é responsável pelo armazenamento dos resultados calculados.

. arquivo **"semantic.c"** : contém as funções responsáveis pela análise semântica do programa "spline" . O módulo **SPLINE** possui uma linguagem de controle de execução; os comandos usados para executar o módulo são validados e interpretados pelas funções deste arquivo. O arquivo é composto por quatro funções : "semantica", "descobre", "erroc" e "inicializa1".

.arquivo **"spline.c"** : contém o programa principal ("main"), a função de cálculo das interpolações ("spline") e mais três funções auxiliares ("rhs", "getlim" e "numb").

Nos itens seguintes são mostrados o procedimento realizado para estruturar o código fonte do módulo **SPLINE** e como medir o número de linhas de código do sistema de software (módulo SPLINE) usando o resultado do número de **seqüências** da **Primeira Forma Normal** de código. São também apresentados os resultados obtidos com a execução do módulo original e do módulo estruturado, sobre o mesmo conjunto de dados, como evidência da eqüivalência funcional entre o código fonte original e o estruturado.

E também realizada uma análise da estrutura original do programa versus a estrutura obtida após a aplicação do **Modelo de Estruturação**; e, mostra-se o tratamento de "código morto" através do resultado obtido com a execução dos protótipos sobre um código fonte fictício.

5.2.1. Estruturação do Módulo SPLINE/NTIA

Para estruturar o módulo devem ser relacionados os seus arquivos fonte e arquivos de inclusão necessários; esta informação pode ser obtida a partir da *"Makefile"* [KERNIGHAN, 84] do módulo, que é a seguinte :

```
# Makefile -------  spline
CFLAGS   = -f -Y -I/usr/users/NTIA/src/libntia -DULTRIX
LDFLAGS  = -s -Y
#SPLINE  = spline.h
#GETOKEN = ../lib/getoken.h
#PCSGBD  = ../lib/pcsgbd.h
#
OBJS  =   spline.o semantic.o lerobs.o getpar.o
          outhdr.o
FNTS  =   spline.c semantic.c lerobs.c getpar.c
          outhdr.c
#
GET   =   sccs get
REL   =
#
spline:         $(OBJS)
                $(CC) $(LDFLAGS) $(OBJS) -lntia -lm -o
                spline
spline.o:       spline.c
semantic.o:     semantic.c
lerobs.o:       lerobs.c
getpar.o:       getpar.c
outhdr.o:       outhdr.c
```

Pela *"Makefile"* tem-se a seguinte relação :

. arquivos fonte : spline.c, semantic.c, lerobs.c, getpar.c e outhdr.c .

. arquivos de inclusão : spline.h e pcsgdb.h.

. diretório dos arquivos de inclusão : ../lib.

A seguir, cada arquivo fonte é pré-processado com o uso do *"cpp"* do UNIX e o seguinte programa *"shell"* :

```
cpp -I../lib spline.c       > spline.cpp
cpp -I../lib semantic.c     > semantic.cpp
cpp -I../lib lerobs.c       > lerobs.cpp
cpp -I../lib getpar.c       > getpar.cpp
cpp -I../lib outhdr.c       > outhdr.cpp
```

No programa acima a opção "-I../lib" especifica o diretório dos arquivos de inclusão para o "cpp".

Para os arquivos gerados pelo programa acima, isto é, arquivos com extensão *".cpp"*, executa-se o protótipo "**C_FNC**", fazendo uso de um segundo programa *"shell"* :

```
c_fnc  <  spline.cpp
c_fnc  <  semantic.cpp
c_fnc  <  lerobs.cpp
c_fnc  <  getpar.cpp
c_fnc  <  outhdr.cpp
```

A seguir, executa-se o protótipo **REDUCAO** para cada **PrimeTree** gerada no processo acima, arquivos com extensão ".**PTF**", especificando-se o terceiro programa *"shell"* :

```
reducao  spline
reducao  rhs
reducao  getlim
reducao  numb
reducao  main
reducao  semantic
reducao  iniciali
reducao  token_va
reducao  descobre
reducao  erroc
reducao  lerobs
reducao  getpar
reducao  number
reducao  usage
reducao  outhdr
```

O código fonte estruturado compõe-se dos arquivos gerados pelo processo acima, que possuem extensão ".FNR".

5.2.2.Número de Linhas de Código do Módulo SPLINE/NTIA

O número de linhas de código de um produto de software é usado como uma medida de representação de tamanho do software, que também é uma estimativa de complexidade [CONTE, 86].

O número de linhas de um código fonte varia de acordo com a linguagem de programação utilizada, dado que para cada caso existem formas diferentes de codificação. Algumas linguagens permitem a composição de comandos em uma mesma linha, ou a continuação do mesmo comando em mais de uma linha, ou as declarações de dados são feitas de formas diferentes [CONTE, 86].

O padrão mais aceito de medida do número de linhas de código é denominado **NLOC** e definido da seguinte forma :

"Uma linha de código é qualquer linha do texto do programa que não é um comentário ou uma linha em branco, independentemente do número de comandos ou fragmentos de comandos na linha. Isto inclui, especificamente, todas as linhas contendo cabeçalhos de declarações, declarações, comandos executáveis e comandos não executáveis do programa." [CONTE, 86]

Ao padronizar o código fonte de um produto de software representado em um conjunto mínimo de subdivisões dos comandos originais, pode-se estabelecer uma medida padrão para o número de linhas de código.

As imposições dos **seqüenciamentos** para obter a **Primeira Forma Normal** correspondem às divisões mínimas dos comandos na base de programas primos adotada; logo, cada **seqüência** corresponde a uma única linha de código.

Assim, foram definidas as seguintes medidas para cada arquivo fonte do software :

. **dcl_glb** : número de linhas de declarações em escopo global às funções do arquivo.

. **dcl_loc1** : número de linhas de declarações locais a uma função, inclusive a declaração da função ("header").

. **n_cmds** : número de comandos em cada função. Foi medido para cada linha terminada em ";", ou contendo "{" ou "}".

. **nloc1** : número de linhas para cada função, correspondendo à soma "dlc_loc1+n_cmds".

. **n_seqs** : número de **seqüências** obtidas na **Primeira Forma Normal** de cada função.

. **dcl_loc2** : número de declarações locais à cada função na **Primeira Forma Normal** (arquivo com extensão ".FNC").

. **nloc2** : número de linhas para cada função na **Primeira Forma Normal**; corresponde à soma "dcl_loc2+n_seqs".

Para o módulo **SPLINE**, de acordo com as medidas acima, tem-se :

Tabela 5.1 : NLOC para as Funções do Arquivo GETPAR.C

	original				padronizado		
funções	dcl_glb	dcl_loc1	n_cmds	nloc1	n_seqs	dcl_loc2	nloc2
----	155	-	-	-	-	-	-
getpar	-	8	88	96	94	19	113
number	-	6	34	40	33	9	42
usage	-	2	4	6	2	3	5
Totais	155	16	126	142	129	31	160

Tabela 5.2 : NLOC para as Funções do Arquivo LEROBS.C

	original				padronizado		
funções	dcl_glb	dcl_loc1	n_cmds	nloc1	n_seqs	dcl_loc2	nloc2
----	137	-	-	-	-	-	-
lerobs	-	1	20	21	27	8	35
Totais	137	1	20	21	27	8	35

Tabela 5.3 : NLOC para as Funções do Arquivo OUTHDR.C

	original				padronizado		
funções	dcl_glb	dcl_loc1	n_cmds	nloc1	n_seqs	dcl_loc2	nloc2
----	137	-	-	-	-	-	-
outhdr	-	4	21	25	26	12	38
Totais	137	4	21	25	26	12	38

Tabela 5.4 : NLOC para as Funções do Arquivo SEMANTIC.C

	original				padronizado		
funções	dcl_glb	dcl_loc1	n_cmds	nloc1	n_seqs	dcl_loc2	nloc2
----	197	-	-	-	-	-	-
semantica	-	2	28	30	30	7	37
token_var	-	6	22	28	34	13	47
descobre	-	1	9	10	11	4	15
erroc	-	3	6	9	6	5	11
inicializa1	-	1	6	7	4	2	6
Totais	197	13	71	84	85	31	116

Tabela 5.5 : NLOC para as Funções do Arquivo SPLINE.C

	original				padronizado		
funções	dcl_glb	dcl_loc1	n_cmds	nloc1	n_seqs	dcl_loc2	nloc2
----	160	-	-	-	-	-	-
rhs	-	3	5	8	6	6	12
spline	-	7	88	95	143	32	175
getlim	-	3	8	11	10	4	14
main	-	6	36	42	39	14	53
numb	-	6	9	15	12	2	14
Totais	160	25	146	171	210	58	268

Deve-se notar que a diferença entre o número de linhas de código para o código original e o código normalizado decorre da padronização adotada.

Para o código original, a medida "n_cmds" foi delimitada por símbolos terminais de comandos (";", "{" e "}"); isto significa que um mesmo comando pode conter várias chamadas a subprogramas em uma mesma expressão, expressões com seleções e qualquer tipo de iteração. No entanto, para o código padronizado cada chamada a subprograma corresponde a uma única **seqüência**; expressões contendo seleções e comandos de iteração são decompostos em mais de uma **seqüência** (vide **seqüenciamentos** do **Apêndice C**).

O número de declarações globais não é afetado, pois qualquer nova declaração de variável, necessária à normalização, é feita no escopo local da função correspondente (vide **seqüenciamentos**, **Apêndice C**). Logo, apenas o número de declarações locais varia. Como a medida é dada para a **Primeira Forma Normal**, é declarada pelo menos uma variável a mais, que é o contador auxiliar "l".

Desta forma, o número de linhas de código (**NLOC**) no padrão adotado pode ser dado pelo número de declarações, globais e locais, mais o número de **seqüências** obtidas pela normalização.

No entanto, deve-se notar que a estimativa é válida para comparar implementações diferentes de um mesmo programa que estejam padronizadas; não é válido comparar as medidas para o módulo **SPLINE** antes e após estruturação, pois o padrão de representação é diferente.

5.2.3. Equivalência Funcional

O **Modelo de Estruturação** aplicado garante que a funcionalidade do código estruturado é equivalente à do código original. Deve-se lembrar que dois programas são funcionalmente equivalentes se para valores de entrada idênticos produzirem valores de saída também idênticos [MARCOTTY, 75].

Assim, para verificar a equivalência funcional entre o módulo **SPLINE** estruturado e o módulo original e também para validar a implementação dos protótipos, os dois módulos foram executados para um mesmo conjunto de dados de entrada.

Os dados referem-se a 36 meses de observação do tempo de trabalho necessário para aquisição da ração essencial mínima, colhidos durante os meses de janeiro a dezembro dos anos de 1979 a 1981 no município de São Paulo (dados utilizados como exemplo no Manual do Módulo SPLINE [NTIA, 92]).

A seguir é apresentado o resultado parcial (o resultado completo encontra-se no **Apêndice D**) de uma interpolação com a execução do módulo original (**SPLINE**) e do módulo **SPLINE** estruturado (denominado "**SPLINES**"), onde os dados originais estão assinalados por "*" :

Tabela 5.6 : Valores Calculados para Interpolação pelo Módulo Original e pelo Módulo Estruturado

Mês	Tempo SPLINE	Tempo SPLINES
* 36.0000	120.4800	120.4800
35.5000	113.1120	113.1120
* 35.0000	119.6800	119.6800
34.5000	146.8829	146.8829
* 34.0000	172.4700	172.4700
33.5000	174.8140	174.8140
* 33.0000	163.7300	163.7300
32.5000	153.3397	153.3397
* 32.0000	143.5500	143.5500
31.5000	132.4596	132.4596
* 31.0000	125.1500	125.1500
30.5000	125.4818	125.4818
* 30.0000	125.4500	125.4500
.	.	.
.	.	.
.	.	.
* 7.0000	138.1800	138.1800
6.5000	135.5835	135.5835
* 6.0000	130.2200	130.2200
5.5000	120.5556	120.5556
* 5.0000	123.7800	123.7800
4.5000	152.1890	152.1890
* 4.0000	179.7800	179.7800
3.5000	180.8257	180.8257
* 3.0000	169.0000	169.0000
2.5000	162.7630	162.7630
* 2.0000	160.3200	160.3200
1.5000	156.2111	156.2111
* 1.0000	150.5700	150.5700

Deve-se observar que para comprovar a equivalência funcional entre os dois módulos é necessário executá-los para todos os possíveis conjuntos de dados de entrada; procedimento que, na prática, é inviável. No entanto, no exemplo acima os resultados produzidos pelos dois módulos são idênticos, o que é uma evidência de que sejam funcionalmente equivalentes.

Logo, a implementação do **Modelo de Estruturação** pode ser considerada satisfatória; e, ainda, para as execuções do módulo original (**SPLINE**) e do estruturado (**SPLINES**) foi gasto o mesmo tempo de CPU para o processo : 0.8 s (em um equipamento de 0.75 MIPS e 6 Megabytes de RAM)

5.2.4 Análise da Estrutura Gerada versus Estrutura Original

O **Modelo de Redução** garante que a variável auxiliar *"1"*, utilizada na estruturação, não é removida do código fonte estruturado se e somente se o código fonte original não é estruturado.

Esta premissa e a análise das funções (**unidades de programa**) do módulo **SPLINES** (**SPLINE** estruturado) permitem concluir que os códigos fonte originais das funções *"getpar"*, *"number"*, *"semantica"* e *"spline"* não são estruturados, pois mantiveram o uso da variável auxiliar *"1"* após o processo de redução. Ou seja, das quinze funções que compõem o módulo, quatro são não estruturadas e onze são estruturadas.

Obtendo os grafos de fluxo de controle para cada função do código fonte original (módulo **SPLINE**) e para cada função estruturada (do módulo **SPLINES**) pode-se calcular a medida ciclomática de McCabe [McCABE, 76] para cada caso.

Os grafos de fluxo de controle foram obtidos com o módulo "chanomat" da ferramenta POKE-TOOL [CHAIM, 91] e a medida ciclomática foi obtida da seguinte forma :

$$V(G) = e - n + 2$$

onde,

e : número de arcos
n : número de nós
G : grafo do fluxo de controle

A padronização imposta pelo **Modelo de Estruturação** pode modificar a estrutura final de um código fonte originalmente estruturado, devido ao subconjunto da linguagem **C** adotado. Nesse caso, os grafos de fluxo de controle e, conseqüentemente, suas medidas ciclomáticas, serão diferentes para o código fonte original e o padronizado. A tabela a seguir evidencia estes resultados :

Tabela 5.7 : Medida Ciclomática
- Funções Estruturadas no Código Fonte Original
- Módulo SPLINE/NTIA

	Original			Estruturado			
	Nós	Arcos	McCabe	Nós	Arcos	McCabe	Diferença*
usage	1	0	1	1	0	1	0
lerobs	13	18	7	16	22	8	1
outhdr	14	19	7	15	21	8	1
token_var	18	25	9	21	30	11	2
descobre	6	7	3	7	8	3	0
erroc	4	4	2	5	6	3	1
inicializa1	1	0	1	1	0	1	0
rhs	2	1	1	5	5	2	1
getlim	8	10	4	9	12	5	1
main	8	10	4	9	11	4	0
numb	6	7	3	8	9	3	0

** Diferença entre a medida ciclomática para o código fonte original e o estruturado.*

Tomemos como exemplo a função *"token_var"*, para a qual o fluxo de controle antes e depois da estruturação é ilustrado pela Figura 5.7; os códigos fonte correspondentes estão no **Apêndice D.**

Pode-se observar na Figura 5.7 que a padronização, obtida com os protótipos, decompôs uma iteração do tipo **"while"** em uma iteração do tipo **"do-while"** e uma seleção do tipo **"if-else"**. Da decomposição resulta o aumento de uma unidade na medida ciclomática de McCabe; como são duas decomposições do mesmo tipo o aumento é de duas unidades.

As funções *"lerobs"*, *"outhdr"*, *"erroc"* e *"getlim"* possuem iterações do tipo **"while"** ou **"for"** no programa original; daí o aumento de uma unidade na medida ciclomática.

No entanto, a função *"rhs"* não possuía iterações do tipo **"for"** ou **"while"**. Para a função *"rhs"* o aumento se deu devido a presença de uma expressão do tipo **" e1 ? e2 : e3 "** (veja **seqüenciamento**, Apêndice C), que é transformada em uma seleção e uma expressão simples; o código fonte original e o estruturado podem ser observados no **Apêndice D.**

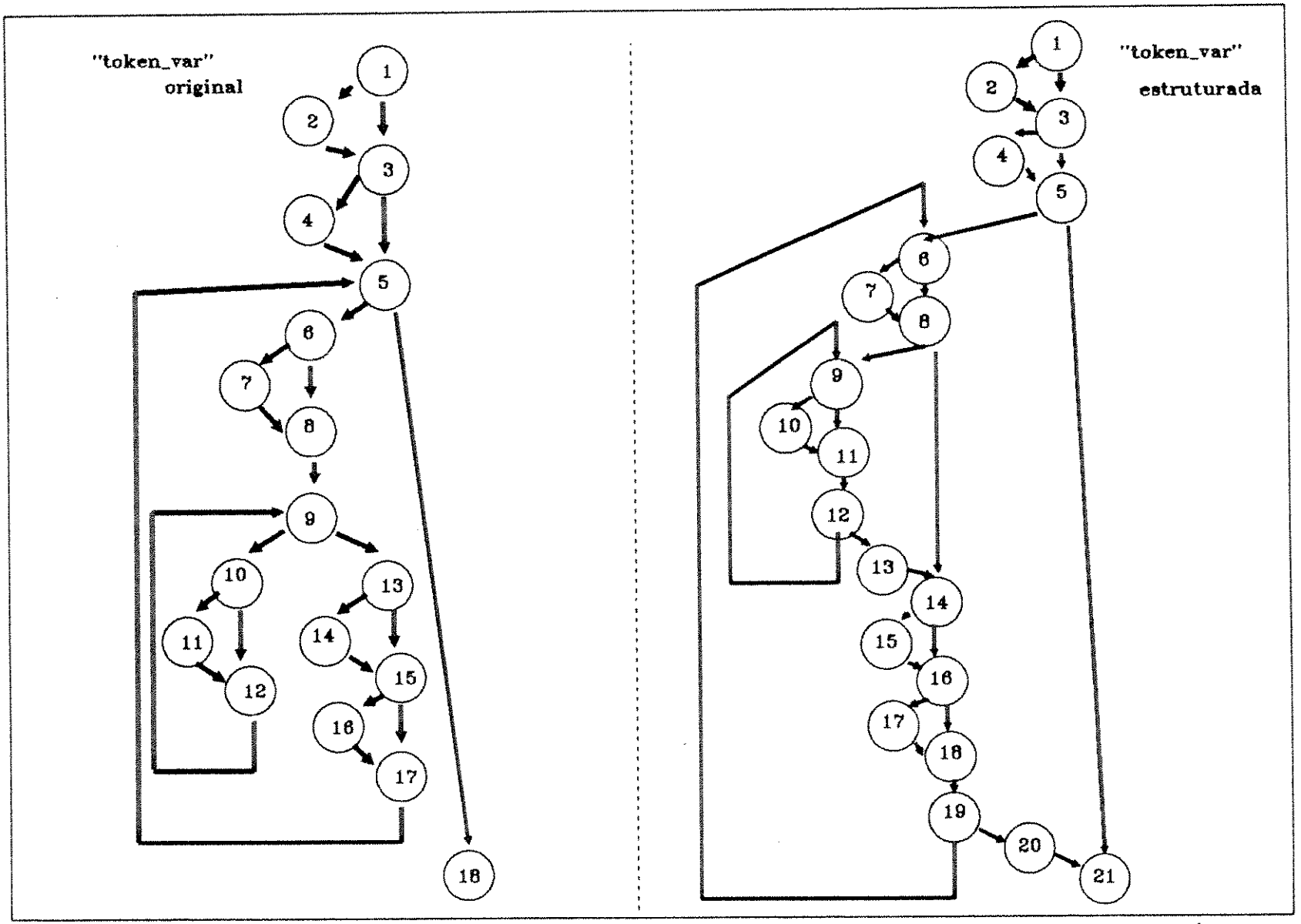

Figura 5.7 : Grafo de Fluxo de Controle – Função "token_var" – SPLINE/NTIA

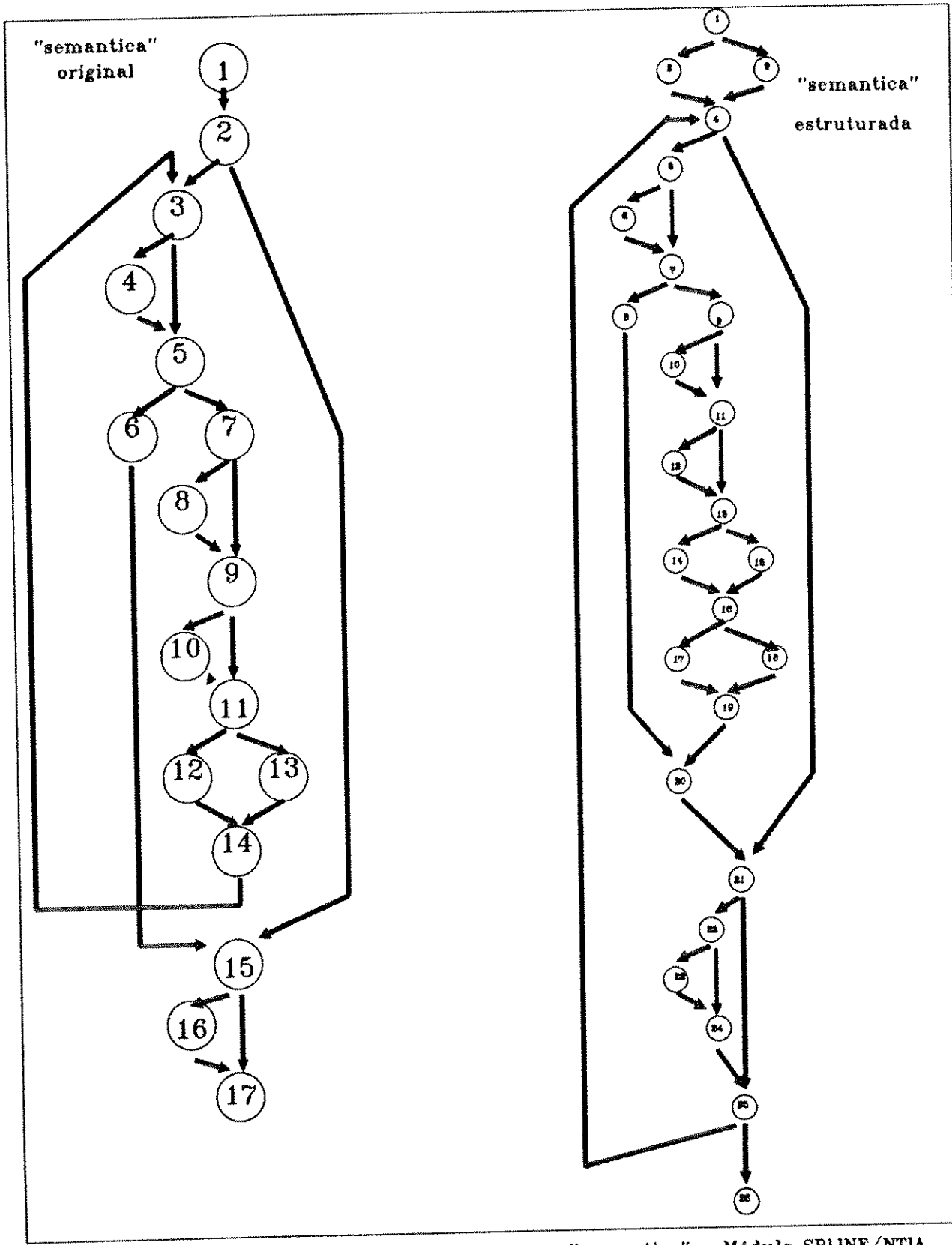

Figura 5.8 : Grafo de Fluxo de Controle - Função "semantica" - Módulo SPLINE/NT1A

Assim, a medida ciclomática de McCabe será a mesma para o código original e para o estruturado se e somente se a estrutura de controle do código fonte original pertencer à base de programas primos imposta. Caso contrário, a variação da medida, $V(G') - V(G)$ aumentará em uma unidade para cada estrutura de controle presente no código fonte original que não pertença ao padrão adotado; G é o fluxo de controle do código fonte original e G' o fluxo de controle do código fonte estruturado.

Para as funções que no código fonte original não são estruturadas, pode-se observar o valor das medidas na seguinte tabela:

Tabela 5.8 : Medida Ciclomática
- Funções Não Estruturadas no Código Fonte Original
- Módulo SPLINE/NTIA

	Original			Estruturado			
	Nós	Arcos	McCabe	Nós	Arcos	McCabe	Diferença*
getpar	44	63	21	70	95	27	6
number	23	35	14	82	112	32	18
semantica	17	23	8	26	36	12	4
spline	47	67	22	85	119	36	14

* *Diferença entre a medida ciclomática para o código original e o padronizado.*

Tomemos como exemplo o grafo de fluxo de controle da função *"semantica"*, antes e depois da estruturação, ilustrados na Figura 5.8 (o código fonte pode ser observado no **Apêndice D**). Pode-se observar que no programa original existia um desvio para fora de um bloco de iteração; este tipo de desvio é considerado uma *forma básica não estruturada* [OULSNAM, 87]. Ao aplicar o **Modelo de Estruturação**, o desvio é mantido dentro do bloco de iteração, assim como o bloco de comandos executado a partir do desvio e da condição final da iteração (Figura 5.8).

A forma básica não estruturada, desvio para fora do bloco de iteração, encobria uma complexidade que estava embebida no programa original [OULSNAM, 79]. Aplicando a medida de McCabe ao fluxo estruturado, o resultado fornece uma estimativa mais confiável da complexidade real do programa.

Logo, a medida sobre o programa estruturado reflete a sua complexidade estrutural real, que poderia estar embutida em uma forma não estruturada [OULSNAM, 79]; por isso, a grande diferença entre as medidas ciclomáticas na tabela anterior.

5.2.5. Tratamento de Código Morto

Consideremos o seguinte exemplo que é um código fonte fictício e válido para a especificação da linguagem C compatível com o PCC :

```
/* morto1.c */
extern    void      abreargs();
extern    void      grava_saida();
extern    int  chave,    qtd;

void morto1() {

     int  i;

     abreargs();
     goto c;
b:   chave = 1;
     qtd += 40;
     goto b;
c:   grava_saida();
     for ( i = 1; i < 20 ; ++i ) {
          printf("codigo nao morto\n");
     }
}
```

Nota-se, no código do exemplo, que o rótulo "*b*" inclui um bloco de três comandos para o qual não existe um caminho executável.

A execução do protótipo **C_FNC** sobre o exemplo gera o arquivo *"morto1.FNC"*, que corresponde ao seguinte programa :

```
/*
   Process : Normalization
   Code Unit : morto1.FNC
   Number of Sequences : 11
   Process Date  : Thu Sep 17 15:09:23 1992
*/
extern int chave , qtd;
extern void abreargs ( ), grava_saida ( );
morto1 ( ){
int  i, FNC_L;
     FNC_L = 1;
     do {
          if (FNC_L==1) {
               abreargs();     FNC_L = 2;
          }
          if (FNC_L==2) {
               FNC_L = 6;
          }
          if (FNC_L==3) {
               chave = 1;      FNC_L = 4;
          }
          if (FNC_L==4) {
               qtd+= 40;       FNC_L = 5;
          }
          if (FNC_L==5) {
               FNC_L = 3;
          }
          if (FNC_L==6) {
               grava_saida(); FNC_L = 7;
          }
          if (FNC_L==7) {
               i = 1;          FNC_L = 8;
          }
          if (FNC_L==8) {
               if (i<20) {
                    FNC_L = 9;
               }
               else {
                    FNC_L = 0;
               }
          }
          if (FNC_L==9) {
               printf("codigo nao morto\n");
               FNC_L = 10;
          }
          if (FNC_L==10) {
               ++i;       FNC_L = 11;
          }
          if (FNC_L==11) {
               if (i<20) {
                    FNC_L = 9;
               }
               else {
                    FNC_L = 0;
               }
          }
       } while (FNC_L > 0) ;
}
```

Embora o código original conte com onze **seqüências**, ou seja, onze comandos executáveis, algumas são **código morto**. As **seqüências** de número 3, 4 e 5 nunca serão executadas pois não existe uma atribuição do tipo *"1=3"* (ou seja, *"FNC_L = 3"*) fora do conjunto de **seqüências** {3,4,5}.

O **Modelo de Redução** identifica este bloco de **seqüências** como código morto e elimina-o do programa, gerando o seguinte resultado :

```
/*
Process : Reduction
Code Unit : morto1.FNR
Process Date  : Thu Sep 17 15:09:56 1992
*/
extern int chave , qtd;
extern void abreargs ( );
extern void grava_saida ( );

void morto1 ( ){

int i;

      abreargs();
      grava_saida();
      i = 1;
      if (i<20) {
            do {
            printf("codigo nao morto\n");
            ++i;
            } while (i<20) ;
      }
}
```

6. CONCLUSOES

6.1 Contribuições do Modelo

Este trabalho constitui o passo inicial de uma abordagem industrial para atacar o problema de reengenharia. A reengenharia é uma das soluções propostas para o problema da "crise de software", pois diminui o custo de manutenção e/ou evolução de sistemas de software e, ainda, busca o reuso de componentes de sistemas de software através da mudança de paradigmas e de plataformas de desenvolvimento.

Para viabilizar a atividade de reengenharia é necessário um processo sistemático e automático de "cleanning-up" do código fonte do produto de software, ou seja, uma padronização de código que leva à estruturação do mesmo - escopo deste trabalho.

As contribuições mais relevantes do **Modelo de Estruturação** proposto, no escopo de reengenharia de códigos fonte de produtos de software implementados em linguagens de programação procedimentais e não paralelas, são apresentadas a seguir :

. **Padronização do código :**

Padronizar as estruturas de controle de qualquer produto de software permite a análise da implementação de diferentes sistemas de software.

A representação normalizada das estruturas de controle, através de uma base mínima de representação - **{ comandos seqüenciais, seleção simples, um único tipo de iteração }**, e a delimitação e reconstrução de **unidades de programa**, permitem obter um ambiente comum para ferramentas de reengenharia e/ou engenharia reversa.

A padronização da representação viabiliza e otimiza processos de remodularização do código [GATTAZ, 84], processos de abstração funcional [EVANGELISTA, 92] e garante a efetividade de medidas estruturais [FENTON, 87].

. **Métrica de complexidade :**

A definição do número de linhas de código (**NLOC**) é quase subjetiva (Seção 5.2.2), dado que depende de como um programa foi escrito em uma determinada linguagem; portanto, não é uma estimativa efetiva que permita comparar implementações diferentes de um mesmo sistema de software.

A primeira etapa do **Modelo de Estruturação**, normalização do código fonte, obtém uma representação padronizada de cada linha de comando do código, através dos **seqüenciamentos** e de acordo com a base de programas primos estabelecida. O resultado desta etapa inclui o número de **seqüências** do código, que fornece uma estimativa efetiva

para o número de linhas de comandos presentes no código padronizado.

Esta normalização aplicada a diferentes sistemas de software, implementados em diferentes linguagens e sob diferentes paradigmas de desenvolvimento, permite compará-los quanto a tamanho e complexidade.

. **Estruturação automática do código fonte :**

O **Modelo de Estruturação** é um processo sistemático, definido por algoritmos, que permite a total automatização do processo. Ou seja, a partir do **Modelo de Estruturação** podem ser obtidas ferramentas automáticas de estruturação de código.

O processo de estruturação não carrega os problemas de limitações da linguagem de programação original, pois leva códigos de uma linguagem procedimental não paralela a uma linguagem estruturada que suporta as estruturas de controle impostas pelo modelo. Os esquemas de decomposição propostos identificam e recompõem **unidades de programa**, garantindo a representação das mesmas em uma linguagem procedimental estruturada.

. **Tradução Fonte a Fonte :**

Os **seqüenciamentos** do código fonte e o **delineamento de unidades de programa** permitem a tradução do código fonte original para uma linguagem de programação estruturada.

O produto final tem sua qualidade garantida, dado que não existe a preocupação em manter as estruturas originais, pois a padronização de estruturas é que leva à tradução, e não vice-versa.

. **Portabilidade do Código Estruturado :**

A linguagem intermediária para representação do código fonte estruturado final, decorrente de cada **seqüenciamento**, é a linguagem **C**, que tem como grande fator de expansão de seu uso a portabilidade - adaptação a diferentes ambientes de produção e desenvolvimento [KERNIGHAN, 88].

6.2 Trabalhos Futuros

O **Modelo de Estruturação** abre um leque de alternativas para evoluções do próprio modelo e implementações de novos processos a partir dele mesmo.

Em primeiro plano é necessário evoluir os protótipos e

implementar o modelo proposto para as linguagens **COBOL** e **FORTRAN**.

A partir das evoluções e novas implementações pode-se experimentar o uso das ferramentas observando o comportamento dos resultados para diferentes sistemas de software. Esta experimentação permitirá identificar novas evoluções para o modelo e ferramentas derivadas.

Um outro interesse é estudar a adaptação do modelo a códigos fonte de produtos de software implementados em linguagens de programação com diferentes modelos computacionais. Este estudo poderia levar a uma evolução do modelo, ou simplesmente a uma nova linguagem intermediária.

O **Modelo de Redução** poderia ser evoluído a partir de um estudo e experimentação de métodos de estruturação sobre grafo de fluxo de controle, obtidos sobre resultados parciais da redução, tendo como objetivo uma forma mais cognitiva para o código final.

Ainda, considerando as necessidades de mercado, poderia ser estudada a possibilidade de construir um estruturador para códigos fonte em linguagem **COBOL** cujo resultado fosse também fornecido em **COBOL**.

Resumidamente pode-se relacionar os seguintes trabalhos futuros :

. **Evoluções para a Implementação do Modelo Proposto :**

. **Evolução dos protótipos :**

Implementar uma interface para os protótipos desenvolvidos e melhorar a forma de gerenciamento de memória (Seção 5.1.1 evoluções necessárias).

. **Implementar o Modelo para COBOL e FORTRAN :**

A implementação do modelo para as linguagens **COBOL** e **FORTRAN** depende da tradução de declarações de estruturas de dados e da construção de bibliotecas de operadores para os nós do tipo **CALL** identificados (vide **seqüenciamentos**, Apêndices **A** e **B**).

. **Resultados Experimentais :**

Estudar o comportamento do **Modelo de Estruturação** aplicado a vários sistemas de software, nas linguagens propostas, e dentro do **Sistema de Reengenharia** (Seção 1.1).

Este estudo visa a validação da suficiência do modelo para o **Sistema de Reengenharia** como um todo, e a análise dos resultados gerados, podendo determinar evoluções necessárias.

. **Evolução do Modelo de Estruturação :**

. **Diferentes Modelos Computacionais :**

O **Modelo de Estruturação** não prevê a aplicação direta a códigos fonte em linguagens de programação com diferentes

modelos computacionais.

O modelo não suporta um operador de sincronização de tarefas, pois foi desenvolvido apenas para linguagens procedimentais sem características explícitas de paralelismo.

A inclusão de um tipo de operador de sincronização na base de estruturas suportadas pelo modelo visa tratar códigos fonte em linguagens que contenham estruturas de controle como corotinas, execução de tarefas em paralelo, esquemas de semáforos e paralelismos diversos.

Com esta evolução o modelo poderia ser aplicado a códigos fonte em linguagens como **ADA**, **C** concorrente, etc; no entanto, também a linguagem de representação final precisaria ser revista.

Códigos fonte em linguagens como SNOBOL4 não podem ser diretamente tratados por este modelo pois, durante a execução de um programa, novos subprogramas e variáveis podem ser criados e toda a estrutura de controle pode ser alterada de acordo com as condições de execução [PRATT, 84].

Logo, o modelo necessita de adaptações para padronizar corretamente códigos em linguagens com paralelismo explícito e variações do programa em tempo de execução.

. Evolução do Modelo de Redução :

O **Modelo de Redução** pode ser evoluído a partir de experimentações de outros algoritmos e métodos de estruturação.

Os três casos seguintes poderiam ser testados :

. Caso 1 :

Utilizar o algoritmo de composição de programas, definido na Seção 2.1.3 como simplificação da **Primeira Forma Normal**.

Neste caso, replicações de trechos de código seriam permitidas e o critério de parada para a composição seria a existência de **seqüências** recursivas. O código final gerado seria "loop-free" (livre de iterações [LINGER, 79]) mas, por outro lado, se existissem iterações, a variável auxiliar "**l**" nunca seria removida do código estruturado e o "loop" externo de execução seria mantido.

. Caso 2 :

Implementar outros modelos de estruturação de programas a partir de um resultado parcial das composições de **seqüências**.

Neste caso, seria utilizado o algoritmo de composição definido no Capítulo 4 (algoritmo denominado

"CompoeElimina"), onde repetições de trechos de código não são permitidas.

Realizando todas as composições possíveis, poderia ser gerado um grafo de fluxo de controle a partir do resultado obtido. Para o grafo de fluxo de controle, cada **seqüência** não composta *"si"* seria o nó de número *"i"* do grafo e cada atribuição do tipo *"l=i"* seria um arco dirigido para o nó *"i"*.

Sobre o grafo de fluxo de controle gerado poderiam ser aplicados outros métodos de estruturação, como os de Oulsnam [OULSNAM, 87] e Ashcroft e Manna [ASHCROFT, 71]. Esses métodos transformam formas não estruturadas a partir de cada predicado que causa desvios, utilizando variáveis auxiliares ligadas aos predicados.

O resultado final produzido possuiria várias variáveis auxiliares; porém, os blocos de iteração seriam reconhecíveis apenas por seus predicativos associados e não por outra variável qualquer.

.Caso 3 :

Manter a implementação do modelo como feita atualmente.

Com as três opções descritas, escolher-se-ia o melhor método a partir de experimentações sobre vários sistemas de software, com respeito ao comportamento para o **Sistema de Reengenharia** (Seção 1.1) e ao estudo dos aspectos cognitivos do resultado obtido.

. **Obter um Estruturador COBOL->COBOL :**

Se a base de programas primos adotada para a normalização for substituída por { **comandos seqüenciais, if-else, perform until** }, pode-se adaptar o modelo para estruturar código fonte em COBOL levando-o a COBOL. Neste caso, os comandos seqüenciais não podem incluir os tratamentos de exceções.

Como o esquema de decomposição delimita as **unidades de código**, cada uma delas pode ser levada a constituir uma nova seção de código ("section"), solução adotada pela ferramenta de estruturação *IBM COBOL Structuring Facility* [LINGER, 88]. Cada composição que formaria um bloco de comandos em uma seleção pode constituir um novo parágrafo; os tratamentos de exceções também podem ser separados em novas seções.

Com as adaptações propostas, o código estruturado produzido pelo modelo pode ser representado em COBOL.

6.3. Considerações Finais :

O **Modelo de Estruturação** proposto pode ser completamente automatizado, fornecendo uma base adequada para um ambiente de reengenharia onde sejam tratados diferentes sistemas de software, implementados em linguagens procedimentais e não paralelas.

Dado que a linguagem de programação do código fonte original seja representável por uma **BNF**, cujo analisador sintático pode ser implementado pelo gerador de "parser" **YACC** do **UNIX**, o modelo pode ser implementado da forma proposta neste trabalho, sob ambiente **UNIX** e para as linguagens **COBOL**, **FORTRAN** e **C**.

As linguagens escolhidas para o desenvolvimento do modelo e para a representação do código estruturado final permitem atingir uma boa parcela dos produtos de software existentes no mercado.

A linguagem **COBOL**, tratada pelo modelo, representa mais de 80 bilhões de linhas de código no mundo [YEH, 90-b]. A linguagem **FORTRAN**, ainda reflete a maior parte das aplicações de processamento científico, e muitos métodos e ferramentas automáticas procuram recuperar estes sistemas e mudar sua plataforma de desenvolvimento. A linguagem **C**, por sua portabilidade e desempenho junto a sistemas "**UNIX-like**", tem tido seu uso aumentado em novas plataformas de desenvolvimento de software e hardware ("workstations", redes, etc).

A utilização de **C** como a linguagem de representação do software estruturado final aumenta a probabilidade de reuso desses sistemas de software, levando-os a uma representação modular e facilitando a portabilidade do sistema de software, devido à tendência crescente de uso dos ambientes "**UNIX-like**".

BIBLIOGRAFIA

[AHO, 86] AHO,A.V. SETHI, R. ULLMAN, J.D. **Compilers : Principles, Techniques and Tools**, Addison-Wesley Pub.Co., 1986.

[AMMARGHELLAT, 92] AMMARGUELLAT, Z. **A Control-Flow Normalization Algorithm and Its Complexity**, IEEE Trans. on Software Engineering, 18(3):237-251, March, 1992.

[ARNOLD, 89] ARNOLD, R.S. **Software Structuring**, Proceedings of the IEEE, 77(4):607-617, April, 1989.

[ASHCROFT, 71] ASHCROFT, E. MANNA, Z. **The Translation of 'go to' Programs to 'while' Programs**, Proc. of the 1971 IFIP Congress, vol. 1, pp.250-255, Amsterdam.

[BAIL, 88] BAIL, W.G. ZELKOWITZ, M.V. **Program Complexity Using Hierarchical Abstract Computers**, Computer Language, Great Britain, 13(3/4):109-123, 1988.

[BAKER, 72] BAKER, B.S. **System Quality Through Structured Programming**, AFIPS Proceedings of the 1972 Fall Joint Computer Conference, vol.41, pp.339-344, 1972.

[BAKER, 77] BAKER, B.S. **An Algorithm for Structuring Flowgraphs**, Journal of the ACM, 24(1):98-120, January, 1977.

[BARTUSSEK, 86] BARTUSSEK, W. PARNAS, D.L. **Using Assertions About Traces to Write Abstract Specifications for Software Modules**, in: Software Specification Techniques, ed.by GEHANI, N. McGETRICK, A.D., Int. Computer Science Series, Addison-Wesley Pub.Co., 1986.

[BASILI, 82] BASILI, V.R. MILLS, H.D. **Understanding and Documenting Programs**, IEEE Transactions on Software Engineering, SE-8(3):270-283, may, 1982.

[BHOM, 66] BHOM, C. JACOPINI, G. **Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules**, Communications from the ACM, 9(5):366-371, May, 1966.

[BYRNE, 91] BYRNE, E.J. **Software Reverse Engineering : A Case Study**, Software - Practice and Experience, 21(12):1349-1364, December, 1991.

[BUSH, 85] BUSH, E. **The Automatic Restructuring of COBOL**, in : Proc. of the Conference on Software Maintenance - 1985 (Washington, DC), IEEE Comp.Soc., pp.35-41, 1985.

[CALDIERA, 90] CALDIERA, G. BASILI, V.R. **Reengineering Existing Software for Reusability**, Technical Report, University of Maryland, MD, USA, UMIACS-TR-90-30, CS-TR-2419, February 1990.

[CALISS, 88] CALISS, F.W. **Problems with Automatic Restructurers**, SIGPLAN Notices, 23(3):13-21, March, 1988.

[CAMARGO, 91] CAMARGO, F.B. **Composição de Componentes de Software : Especificação Formal de Requisitos e Automatização do Processo de Validação**, Tese de Mestrado, DCA/FEE/UNICAMP, Dezembro de 1991.

[CANTONE, 86] CANTONE, G. CIMITILE, A. DE CARLINI, U. **Well-formed Conversion of Unstrucutured One-in/one-out Schemes for Complexity Measurement and Program Maintenance**, The Computer Journal, 29(4):322-329, 1986.

[CARNASSALE, 91] CARNASSALE, M. **GFC - Uma Ferramenta Multi-Linguagem para Geração de Grafo de Programa**, Tese de Mestrado, DCA/FEE/UNICAMP, Fevereiro de 1991.

[CHAIM, 91] CHAIM, M.L. **POKE-TOOL - Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados**, Tese de Mestrado, DCA/FEE/UNICAMP, Abril de 1991.

[CHAIM, 92] CHAIM, M.L. **COB_FNC : Um Tradutor Fonte a Fonte COBOL - Primeira Forma Normal de Código**, NTIA /EMBRAPA (Relatório Interno), 1992.

[CHIKOFSKY, 90] CHIKOFSKY, E.J. CROSS, J.H. **Reverse Engineering and Design Recovery : A Taxonomy**, IEEE Software, January, 1990.

[CONTE, 86] CONTE, S.D. DUNSMORE, H.E. SHEN, V.Y. **Software Engineering Metrics and Models**, The Benjamin/Cummings Pub. Co., 1986.

[COWELL, 79] COWELL, D.F. GILLIEST, D.F. KAPOSI, A.A. **Synthesis and Structural Analysis of Abstract Programs**, The Computer Journal, 23(3):243-247.

[CREAK, 87] CREAK, G.A. **When GOTO goes to, how does it get there?**, SIGPLAN Notices, 22(2):36-42, February, 1987.

[DEC-FORTRAN, 75] **DEC-System10 FORTRAN IV (F40) Programmer´s Reference Manual**, Order DEC-10-LFLMA-B-D, DEC, Maynard, Massachussets, 1975.

[DIJKSTRA, 79] DJIKSTRA, E.W. **GO TO Statement Considered Harmful**, Comm. of The ACM, 11(3):147:148, March, 1978. Also in : Classics of Software Engineering, Edward Nash Yourdon, 1979, New York.

[ELSHOFF, 78] ELSOFF, J.L. MARCOTTY, M. **On the Use of the Cyclomatic Number to Measure Program Complexity**, SIGPLAN Notices, 13(12):29-40, December, 1978.

[EVANGELISTA, 92] EVANGELISTA, S.R.M. **Abstração Funcional através de Segmentação de Programas e Composição Funcional**, Tese de Mestrado, DCA/FEE/UNICAMP, Novembro de 1992.

[FELDMAN, 90] FELDMAN, S.I. GAY, D.M. MAIMONE, M.W. SCHRYER, N.L. **A FORTRAN-to-C Converter**, AT&T Bell Laboratories, Computing Science Technical Report n. 149, Murray Hill, N.J., June, 1990.

[FENTON, 86] FENTON, N.E. WHITTY, R.W. **Axiomatic Approach to Software Metrication through Program Decomposition**, The Computer Journal, 29(4):330-339, 1986.

[FENTON, 87] FENTON, N.E. KAPOSI, A.A. **Metrics and Software Structure**, Information and Software Technology, London, 29(6):301-320, July/August, 1987.

[GANNON, 83] GANNON, J.D. HECHT, M.S. HERBOLD, R.J. **Prime Program Decomposition**, Proceedings of the Sixteenth Annual Hawaii International Conference on System Sciences, pp. 25-29, 1983.

[GATTAZ, 82] GATTAZ, F.Sbr. **Systematic Elimination of GOTO Statements from COBOL Programs**, Technical Report, TR-I101/82-0096-2, International Software Systems Inc., College Park, MD, 1982.

[GATTAZ, 84] GATTAZ, F.Sbr. **Structural Complexity : a Basis for Systematic Software Evolution**, Ph.D. Dissertation, University of Maryland, MD, USA, 1984.

[HAUSLER, 90] P.A. PLESZKOCH, M.G. LINGER, R.C. HEVNER, A.R. **Using Function Abstraction to Understand Program Behavior**, IEEE Software, January, 1990.

[HOPKINS, 72] HOPKINS, **A Case for the GOTO**, SIGPLAN Notices, 7(11):59-62, November, 1972.

[HEHL, 79] HEHL, M.E. **Sistema de Programação FORTRAN IV GH**, Mc-Graw-Hill do Brasil Ltda., 1979.

[HOPCROFT, 79] HOPCROFT, J.E. ULLMAN, J.D. **Introduction to Automata Theory, Languages and Computation**, Addison-Wesley Pub.Co., 1979.

[IBM, 83] Curso de COBOL da IBM, COBOL-ANS - Codificação de Cartão e Aplicação em Fita Magnética, Instrução Programada, Manual n. SR 17-0283-2; e Codificação COBOL-ANS - Ilustrações, Instrução Programada, Manual n. SR 17-0285-1; Rio de Janeiro, R.J., 1983.

[JOHNSON, 84-a] JOHNSON, S.C. **A Tour Through the Portable C Compiler**, ULTRIX-32™ Supplementary Documents, Vol. II - Programmers, Order AA-BG67A-TE, DEC, Merrimack, New Hampshire, 1984.

[JOHSON, 84-b] JOHNSON, S.C. **YACC : Yet Another Compiler-Compiler**, ULTRIX-32™ Supplementary Documents, Vol. II - Programmers, Order AA-BG67A-TE, DEC, Merrimack, New Hampshire, 1984.

[KERNIGHAN, 84] KERNIGHAN, B.W. PIKE, R. **The UNIX Programming Environment**, Prentice-Hall Inc., 1984.

[KERNIGHAN, 86] KERNIGHAN, B.W. **C - A Linguagem de Programação**, 4. edição, Editora Campus LTDA, 1986.

[KNUTH, 77] KNUTH, D.E. **Structured Programming with GO TO Statements**, in : Current Trends in Programming Methodology, Vol. I, R.T.Yeh, ed., 1977.

[LEHMANN, 80] LEHMAN, M.M. **Programs, Life Cycles and Laws of Software Evolution**, Proceedings of the IEEE, 68(9):199-215, September, 1980.

[LEITAO, 92] LEITAO, P. S. Jr. **Suporte ao Teste de Programas COBOL no Ambiente POKE-TOOL**, Tese de Mestrado, DCA/FEE/UNICAMP, Agosto de 1992.

[LESK, 84] LESK, M.E. SCHIMIDT, E. **Lex - A Lexical Analyzer Generator**, ULTRIX-32™ Supplementary Documents, Vol. II - Programmers, Order AA-BG67A-TE, DEC, Merrimack, New Hampshire, 1984.

[LINGER, 79] LINGER, R.C. MILLS, H.D. WITT, B.I. **Structured Programming : Theory and Practice**, Addison-Wesley Pub.Co. Inc., 1979.

[LINGER, 88] LINGER, R.C. MILLS, H.D. **A Case Study in Cleanroom Software Engineering : The IBM COBOL Structuring Facility**, in: Proc. COMPSAC 1988, IEEE Comp.Soc., 1988.

[MACARIO, 91] MACARIO, C.G. CAMARGO, F.B. MOURA, M.F. EVANGELISTA, S.R.M. TERNES, S. **Metodologia de Componentização de Sistemas de Software**, Plano de Trabalho 1991/1992, NTIA/EMBRAPA (Relatório Interno), Setembro, 1991.

[MACARIO, 92] MACARIO, C.G. CAMARGO, F.B. CHAIM, M.L. MOURA, M.F. REZENDE, M. HIGA, R. EVANGELISTA, S.R.M. TERNES, S. **Projeto de Reengenharia**, in: Plano Diretor da Unidade, NTIA/EMBRAPA (Relatório Interno), Abril, 1992.

[MARCOTTY, 75] MARCOTTY, M. LEDGARD, H.F. **A Genealogy of Control Structures**, Communications of the ACM, 18(11):629-639, November, 1975.

[McCABE, 76] McCABE, T.J. **A Complexity Measure**, IEEE Transactions on Software Engineering, SE-2(4):3-15, December, 1976.

[MICROSOFT, 80] MICROSOFT COBOL Programming Language, COBOL-80, Reference Manual MMIC 4399, n. 83 01-401-02, Microsoft, 1980.

[MILLER, 87] MILLER, J.C. STRAUSS III, B.M. **Implications of Automated Restructuring of COBOL**, SIGPLAN Notices, 22(6):76-82, June, 1987.

[MILLER, 90] MILLER, J.C. **Reengenharia de Software : Aprontar é Duas Vezes Mais Divertido**, in : Reengenharia de Software - Técnicas de MAnutenção de Programas e Sistemas, PARIKH, G., Livros Técnicos e Científicos Ed., 1990.

[MILLER, 91] MILLER, J.C. **Software Re-engineering - An Introduction**, s.n.t., s.l., IBPI, s.d., pp. 61. (Seminário, 1991, São Paulo, S.P.)

[MORGAN, 84] MORGAN, H.M. **Evolution of a Software Maintenance Tool**, Proc. of the 2nd National Conference on EDP Software Maintenance, Silver Spring, MD, U.S. Professional Dev. Institute, pp.268-278, 1984.

[MOURA, 91] MOURA, M.F. **Reestruturação de Programas**, Plano Anual de Trabalho 91/92, NTIA/EMBRAPA (Relatório Interno), 1991.

[NTIA, 92] NTIA/EMBRAPA **Software NTIA - Manual do Usuário**, versão 92, NTIA/EMBRAPA, 1992.

[OULSNAM, 79] OULSNAM, G. **Cyclomatic Numbers do not Measure Complexity of Unstructured Programs**, Information Processing Letters, 9(5):207-211, December, 1979.

[OULSNAM, 82] OULSNAM, G. **Unravelling Unstructured Programs**, The Computer Journal, 25(3):379-387, 1982.

[OULSNAM, 87] OULSNAM, G. **The Algorithmic Transformation of Schemas to Structured Form**, The Computer Journal, 30(1):43-51, 1987.

[PARNAS, 72] PARNAS, D.L. **On the Criteria to Be Used in Decomposign Systems into Modules**, Communications of The ACM, 5(12):1053-1058, 1972.

[PRATT, 84] PRATT, T.W. **Programming Languages Design and Implementation**, Prentice-Hall, Inc, Englewood Cliffs, New Jersey, Second Edition, 1984.

[RAMSHAW, 88] RAMSHAW, L. **Eliminating go to's while Preserving Program Structure**, Journal of the ACM, 35(4):893-920, October, 1988.

[REZENDE, 92-a] REZENDE, M. **Descrição do Algoritmo SLICER**, NTIA/EMBRAPA (Relatório Interno), 1992.

[REZENDE, 92-b] REZENDE, M. **Descrição do Algoritmo TRACER**, NTIA/EMBRAPA (Relatório Interno), 1992.

[TERNES, 92] TERNES, S. **Medidas de Complexidade para Avaliação de Mudanças em Sistemas de Software**, Tese de Mestrado, DCA/FEE/UNICAMP, Junho, 1992.

[TSE, 87] TSE, T.H. **The Identification of Program Unstructuredness : A Formal Approach**, The Computer Journal, 30(6):507-511, 1987.

[YEH, 90-a] YEH, R.T. **An Alternative Paradigm for Software Evolution**, In: NG, P.A. & YEH, R.T. Modern Software Engineering, New York, Van Nostrand Reinold, 1989, p.7-22.

[YEH, 90-b] YEH, R.T. NG, P.A. GATTAZ, F.Sbr. FERRARETO, M.D. Executive Summary, **Proc. of 2nd International Workshop on the SPP**, Software Plant Project, Campinas, SP, Brazil, March 6th to 8th, 1990.

[WIRTH, 74] WIRTH, N. **On the Composition of Well-Structured Programs**, ACM Computing Surveys, 6(4):247-259, December, 1974.

[WULF, 72] WULF, W.A. **A Case Against The GOTO**, Proc. of the 25th National ACM Conference, August, 1979, pp. 791-797.

[URSCHLER, 75] URSCHLER, G. **Automatic Restructuring of Programs**, IBM J.Res.Dev., vol.19, pp.181-194.

APENDICE A

Seqüenciamentos para as Estruturas de Controle da Linguagem COBOL

Neste apêndice são relacionadas as principais estruturas de controle da linguagem COBOL e seus **seqüenciamentos**. A relação de estruturas de controle foi obtida do curso de **COBOL** ANS da IBM [IBM, 83] e do manual da Microsoft [MICROSOFT, 80]; os **seqüenciamentos** foram definidos de acordo com as estruturas correspondentes apresentadas na **Tabela de Transformações (Tabela 2.2.)**.

1. **Composição :** um conjunto de comandos com execução seqüencial é uma composição.

Exemplo :

Para a seguinte composição :

```
MOVE 1 TO B.
ADD  2 TO A.
WRITE LINHA AFTER 2 LINES.
```

Temos o seguinte seqüenciamento :

```
i   : B = 1;                        l = i + 1;
i+1 : A = A + 2;                    l = i + 2;
i+2 : write(linha, after, 2);       l = i + 3;
```

Note-se que o comando "**write**" foi tratado como uma função. O mesmo tratamento é aplicado a qualquer comando específico de entrada e saída, ou comandos da linguagem com funções embutidas, como os comandos "**SORT**", "**SEARCH**", etc.

A linguagem de programação **COBOL** apresenta várias exceções; por exemplo, a grande quantidade de cláusulas opcionais que podem ser usadas em comandos simples, como um "**ADD**" ou um "**MOVE**". Logo, nem sempre um comando apresenta uma única funcionalidade.

Por exemplo :

```
ADD a1,a2,..,an ROUNDED ON SIZE ERROR GO fim
OTHERWISE MOVE OK TO calculo.
```

Essa estrutura em um comando "**ADD**" gera uma composição de comandos que nem sempre tem execução seqüencial; por exemplo, no caso acima há um "**GO**" associado.

Uma alternativa é decompor o comando da seguinte forma :

```
i      : an = a1 + a2 + ... ;              l = i+1
i+1    : c = rounded(an,SIZE ERROR);       l = i+2
i + 2  : if ( c )                          l = i+3
           cc                              l = i+4
i + 3  :                                   l = <fim>
i + 4  : calculo = OK ;                    l = i+ 5
```

Onde, *l = <fim>* deverá ser substituído por *l = <número da seqüência do rótulo fim>*; no entanto, isto só poderá ser feito quando o rótulo *"fim"* for encontrado no programa, da mesma forma que no tratamento de desvios (vide **Tabela de Transformações**).

Assim como as funções das cláusulas *"rounded"* e *"size error"* são substituídas por uma função que identifica a ocorrência de *"overflow"*, o mesmo artifício pode ser empregado a outras cláusulas tais como *"invalid key"*, *"at end"*, *"corresponding"*, etc.

2. **Seleção** : vários comandos em COBOL levam à presença de seleções na estrutura de controle devido às cláusulas opcionais e/ou de tratamentos de exceções.

Na linguagem temos o comando *"IF"* com ou sem *"ELSE"*, cujo seqüenciamento é feito da seguinte forma :

Comando original :

```
if < condição >  <stm1>
else             <stm2>.
```

Seqüenciamento :

```
i      : if <condição>     l = i + 1
         else              l = i + 3
i + 1  : <stm1> ;          l = i + 2
i + 2  :                   l = i + 4
i + 3  : <stm2> ;          l = i + 4
i + 4  :
```

O comando *"GO"* com a cláusula *"depending on"*, tem o mesmo tipo de seqüenciamento de uma seleção múltipla.

Exemplo :

```
GO   A, B, C depending on paraonde.
...
```

Seqüenciamento :

```
1     : if (paraonde == 1 )   l = i + 1;
        else                  l = i + 2;
i + 1 :                       l = <A>;
i + 2 : if ( paraonde == 2 )  l = i + 3;
        else                  l = i + 4;
i + 3 :                       l = <B>;
i + 4 : if (paraonde == 3 )   l = i + 5;
        else                  l = i + 6;
i + 5 :                       l = <C>;
i + 6 : ...
```

Comandos com cláusulas condicionais também podem ser classificados como seleções como : o comando *"ADD"* com *"SIZE ERROR"*, o comando *"READ"* com *"AT END"*, etc.

3. **Iterações** : iterações em **COBOL** são realizadas através do uso do comando *"PERFORM"*; o comando é tratado como um nó do tipo **CALL**, dado que o efeito no fluxo de controle é o mesmo que o de uma chamada a subprograma. No entanto, algumas transformações devem ser observadas para preservar a funcionalidade do "**loop**" (laço).

Exemplo :
perform LeituraDeDados until FimArquivo.

Seqüenciamento :

```
1 : if (  FimArquivo )   l = 4;
    else                 l = 2;
2 : LeituraDeDados();    l = 3;
3 : if (  FimArquivo )   l = 4;
    else                 l = 2;
```

No **seqüenciamento** acima a condição é testada antes da chamada da função, o que pode ser entendido como uma iteração do tipo "**while**"; logo, o comando é decomposto da mesma forma (vide **Tabela de Transformações**).

O comando "**perform**" com a cláusula "**times**" é seqüenciado de forma semelhante.

Exemplo :
perform paragrafo n times.

Seqüenciamento :

```
1 : if ( n > 0 )          l = 2;
    else                  l = 6;
2 : AuxVezes = 0;         l = 3;
3 : paragrafo();          l = 4;
4 : ++AuxVezes ;          l = 5;
5 : if ( AuxVezes < n )   l = 3;
    else                  l = 6;
```

Note-se que para o **seqüenciamento** acima foi necessário criar uma variável auxiliar que não interfere sobre o valor da variável "**n**" e possibilita o controle da iteração.

4. Desvios : Os desvios na linguagem COBOL são representados pelo verbo *"GO"* e direcionados a um rótulo especificado no programa. Existe desvio direto, com o verbo e um rótulo, e desvio condicionado, com a cláusula *"depending on"*. O comando *"ALTER"*, no entanto, pode mudar o destino de um ou mais desvios.

Por exemplo, o seguinte trecho de programa :

```
A.
     if condicao1   GO  Rotulo1.
B.
     Move 1 to C.
Rotulo1.
     perform facaoutracoisa.
     GO A B depending on condicao2.
```

Tem o seguinte seqüenciamento :

```
1 : if condicao1                1 = 2;
     else                       1 = 3;
2 :                             1 = 4;
3 : c = 1;                      1 = 4;
4 : facaoutracoisa();           1 = 5;
5 : if ( condicao2 == 1 )       1 = 6;
     else                       1 = 7;
6 :                             1 = 1;
7 : if (condicao2 == 2 )        1 = 8;
     else                       1 = 9;
8 :                             1 = 3;
```

No entanto, se um comando *"ALTER"* está presente no programa, ele é tratado pelo **seqüenciamento** como uma cláusula associada ao verbo *"GO"*, ao qual o *"ALTER"* se referencia.

O programa é seqüenciado até que um *"ALTER"* seja encontrado. Quando o *"ALTER"* é encontrado, todo o comando *"GO"* que envolva os parágrafos que o *"ALTER"* referencia são trocados por nós do tipo OR; como um *"GO"* com cláusula *"depending on"*.

Exemplo :

```
chave.
     perform procedimento1.
aberta.
     perform procedimento2.
     if condicao
          go chave.
fechada.
     perform procedimento3.
saida.
     ALTER chave to fechada.
     GO aberta.
```

Seqüenciamento :

```
1 : procedimento1();        1 = 2;
2 : procedimento2();        1 = 3;
3 : if (condicao)           1 = 4;
    else                    1 = 5;
4 :                         1 = 1;
5 : procedimento3();        1 = 6;
6 : alter 1 to 5;           1 = 7;
7 :                         1 = 2;
```

Quando a **seqüência** 6 é encontrada, os *"GO"*'s encontrados anteriormente referentes à **seqüência** de número 1 devem possuir uma cláusula associada, sendo trocados por uma **seleção** (nó do tipo **OR** na **PrimeTree**); o comando *"ALTER"* passará a ser o comando que controla o valor da condição, como mostrado a seguir:

```
1 : alter = 0;              1 = 2;
2 : procedimento1();        1 = 3;
3 : procedimento2();        1 = 4;
4 : if (condicao)           1 = 5;
    else                    1 = 6;
5 : if ( ! alter )          1 = 2;
    else                    1 = 6;
6 : procedimento3();        1 = 7;
7 : alter=1;                1 = 8;
8 :                         1 = 3;
```

5. **Subprogramas** : Como mostrado nos exemplos anteriores todo comando *"PERFORM"* é tratado como uma chamada de subprograma.

Exemplo :

```
perform paragrafo.
move 2 to variavel.
```

Seqüenciamento :

```
1 : paragrafo();     1 = 2;
2 : variavel = 2;    1 = 3;
```

A cada subprograma corresponde uma **PrimeTree** que representa a **Primeira Forma Normal** do subprograma; assim, para um comando *"perform"* deve ser criada uma **PrimeTree** para o parágrafo e/ou seção por ele referenciado.

Se o comando *"perform"* contém a cláusula *"through"*, então o conjunto de comandos envolvidos nos parágrafos ou seções delineados pela cláusula formam uma única **PrimeTree**; o comando *"perform <par1> through <par2>"* é transformado em um nó do tipo **CALL** : *"<par1>_<par2>();"*.

Exemplo :

```
a.
     perform c through e.
b.
     <conjunto de b comandos em "b">.
c.
     <conjunto de c comandos em "c">.
d.
     <conjunto de d comandos em "d">.
e.
     <conjunto de e comandos em "e">.
```

O seqüenciamento do programa é aplicado normalmente, a diferença ocorre no "**esquema de decomposição**" (apresentado na seção 3.1). Assim o seqüenciamento é o seguinte :

```
i     : c_e();
i + 1 : <conjunto de b comandos em "b">
i + b : <conjunto de c comandos em "c">
i + c : <conjunto de d comandos em "d">
i + d : <conjunto de e comandos em "e">
```

APENDICE B

Seqüenciamentos para as Estruturas de Controle da Linguagem FORTRAN

Neste apêndice são relacionadas as principais estruturas de controle da linguagem **FORTRAN** e seus **seqüenciamentos**. A relação de estruturas de controle para o FORTRAN 77 foi obtida no livro "Programming Languages - Design and Implementation" [PRATT, 84]; para o FORTRAN IV no livro de sistemas de programação em **FORTRAN IV** [HEHL, 79] e no manual de **FORTRAN IV** da DEC [DEC FORTRAN, 75]. Os **seqüenciamentos** foram estabelecidos de acordo com a **Tabela 2.2 (Tabela de Transformações)**.

1. **Composição** : um conjunto de comandos com execução seqüencial é uma composição.

Para a seguinte composição :

```
      B = 1
      A = 2
      PRINT 20, LINHA
20    FORMAT(1H0,1H0,A80)
```

Tem-se o seguinte seqüenciamento :

```
i    : B = 1;                      l = i + 1;
i+1  : A = A + 2;                  l = i + 2;
i+2  : print(LINHA, formato);      l = i + 3;
```

Deve-se notar que o comando "**print**" é tratado como um subprograma, o mesmo tratamento é dado a qualquer comando específico de entrada e saída (como os comandos : "**write**", "**read**", "**type**", "**reread**", "**accept**", "**pause**", etc). Os formatos (comando "**FORMAT**") de entrada e saída de dados são tratados como parâmetros das funções de entrada e saída.

Outro exemplo de composições são expressões da linguagem que envolvam chamadas de funções. Isto porque, uma chamada de função deve corresponder a uma **seqüência**, com um único nó do tipo **CALL**, e o restante da expressão deve estar presente nas **seqüências** sucessoras.

Exemplo :

```
A = (Q+V(M,MAX0(A,B))*Y**2)/(G*H - F(K+3))
```

Seqüenciamento :

```
i     :   f1 = max0(a,b);
          l = i + 1;
i + 1 :   f2 = exp(y,2);
          l = i + 2;
i + 2 :   a = (q+v[m,f1])*f2)/(g*h-f[k+3]);
          l = i + 3;
```

2. Seleção : a linguagem possui pelo menos dois tipos de comandos de seleção : um "if" aritmético e um "if" lógico. Algumas novas versões, como FORTRAN 77, possuem "if-then-else".

A seguir, são mostradas as decomposições, ou seja, os seqüenciamentos, para cada caso:

a. "IF-THEN-ELSE"

Exemplo :

```
if < condição >      <stm1>
else                 <stm2>
```

Seqüenciamento :

```
i      : if <condição>     l = i + 1;
         else              l = i + 3;
i + 1  : <stm1> ;          l = i + 2;
i + 2  :                   l = i + 4;
i + 3  : <stm2> ;          l = i + 4;
i + 4  : ...
```

b. "IF" aritmético

Exemplo :

```
IF (A)  <n1>,<n2>,<n3>
```

Seqüenciamento :

```
i      : if ( a < 0 )          l = i + 1;
         else                  l = i + 2;
i + 1  :                       l =  <n1>;
i + 2  : if ( a==0)            l = i + 3;
         else                  l = i + 4;
i + 3  :                       l =  <n2>;
i + 4  : if ( a > 0 )          l = i + 5;
          else                 l = i + 6;
i + 5  :                       l =  <n3>;
i + 6  : ...
```

onde,
<nj> , j = 1,2,3 : número da seqüência correspondente ao rótulo nj.

c. "IF" lógico

Exemplo :
IF (<expressão booleana>) <comando FORTRAN>

Seqüenciamento :

```
i     : if (<expressão booleana>) l = i + 1;
        else                      l = i + 2;
i + 1: <comando FORTRAN decomposto>; l=i+2;
```

3. **Iterações** : iterações em **FORTRAN** são realizadas pelo comando "DO"; ou simulando-se um bloco de iteração com a utilização de "**if´s**" e "**goto´s**".

No caso da simulação de blocos de iteração, as decomposições são aplicadas diretamente, como para os desvios apresentados na **Tabela de Transformações.**

Exemplo :

```
     I = 1
40   IF ( I.EQ.N )        GOTO  50
     CALL PARAGRAFO()
     I = I + 1
     GOTO  40
50   ...
```

Seqüenciamento :

```
1 :  I = 1;                  1 = 2;
2 :  if ( I == N )           1 = 3;
     else                    1 = 5;
3 :                          1 = 7;
4 : PARAGRAFO();             1 = 5;
5 : I = I + 1;               1 = 6;
6 :                          1 = 2;
7 : ...
```

Para o comando "DO" o **seqüenciamento** depende do significado semântico do comando.

No **FORTRAN IV** [HEHL, 79] o bloco de comandos interno a um "DO" é executado pelo menos uma vez, dado que o teste de parada só é realizado ao final do bloco; da mesma forma que em uma iteração do tipo "**DO-WHILE**" da **Tabela de Transformações.** Neste caso, temos :

Exemplo : (DO para FORTRAN IV)

```
     DO  50  K=J, N, 1
     M(K) = M(K) * 2
50   CONTINUE
     ...
```

Seqüenciamento :

```
i     : K = J;               1 = i+ 1;
i + 1 : M[K] = M[K] * 2;     1 = i+ 2;
i + 2 : K = K + 1;           1 = i+ 3;
i + 3 : if ( K <= N )        1 = i+ 1;
        else                 1 = i+ 4;
i + 4 : ...
```

O mesmo comando para o **FORTRAN 77** tem outra execução. Seu fluxo de controle é o mesmo de uma iteração do tipo "**WHILE**", **Tabela de Transformações do Capítulo 2.**, ou seja, o teste de parada é realizado antes da execução do bloco.

Exemplo :

```
     DO  50  K=J, N, 1
     M(K) = M(K) * 2
50   CONTINUE
     ...
```

Seqüenciamento :

```
i     : K = J;              1 =i + 1;
i + 1 : if ( K <= N )       1 =i + 2;
        else                1 =i + 5;
i + 2 : M[K] = M[K] * 2;    1 =i + 3;
i + 3 : K = K + 1;          1 =i + 4;
i + 4 : if ( K <= N )       1 =i + 2;
        else                1 =i + 5;
i + 5 : ...
```

Todo **seqüenciamento** a ser aplicado a uma estrutura depende de seu fluxo de controle. Nos dois casos anteriores a variação deve-se a particularidade das versões da linguagem **FORTRAN** e ao **seqüenciamento** empregado sempre associar um "**do-while**" à estrutura de um "**loop**" (devido ao sub-conjunto da linguagem **C** utilizado).

4. Desvios : Os desvios na linguagem **FORTRAN** são representados pelo comando "**GOTO**" e direcionados a um rótulo especificado no programa.

Existem três tipos básicos de "GOTO" :

a. Desvio simples para um único rótulo

Exemplo :

GO TO N

Seqüenciamento :

1 : 1 = <N>

onde,

<N> : número da seqüência à qual refere-se o rótulo N.

b. "GOTO" envolvendo o cálculo de um rótulo pré-definido

Tem o seqüenciamento de uma seleção múltipla.

Exemplo :

```
GO  TO (<ra>, <rb>, <rc>), paraonde
...
```

Seqüenciamento :

```
i     : if ( paraonde == 1 )   l = i + 1;
         else                  l = i + 2;
i + 1 :                        l = <ra>;
i + 2 : if ( paraonde == 2 )   l = i + 3;
         else                  l = i + 4;
i + 3 :                        l = <rb>;
i + 4 : if ( paraonde == 3 )   l = i + 5;
         else                  l = i + 6;
i + 5 :                        l = <rc>;
i + 6 : ...
```

onde ,

<rx>, x = a,b,c : número da seqüência correspondente ao rótulo <rx>.

c. "GOTO" assinalado

E um comando "GOTO" que pode ser alterado por um comando "ASSIGN". O seqüenciamento do GOTO é simples pois as possibilidades de alteração estão presentes no próprio comando.

Exemplo :

```
...
ASSIGN S TO M
...
GO TO M,(S1,S2,...,Sn)
...
```

onde,

M : valor que corresponde ao rótulo para onde o programa deve ser desviado.
S_j, j = 1,..,n : valores que podem ser assinalados para M.
S : valor assinalado para M no comando "ASSIGN".

Seqüenciamento :

```
i          : ...
i + 1      : M = S;              l = i + 2;
             ...
k          : if ( M == S1 )      l = k + 1;
             else                l = k + 2;
k + 1      :                     l = <S1>;
k + 2      : if ( M == S1 )      l = k + 3;
             else                l = k + 4;
k + 3      :                     l = <S2>;
             ...
k + n - 1  : if ( M == Sn )      l = k + n;
             else                l = k + n + 1;
k + n      :                     l = <Sn>;
k + n + 1  : ...
```

onde,

<S_j>, $j = 1,..,n$: número da seqüência à qual corresponde o rótulo <S_j>.

5. Subprogramas : A linguagem prevê a definição de três tipos de subprogramas, cujos exemplos e seqüenciamentos são mostrados a seguir.

a. subroutines

As chamadas são feitas exclusivamente com o uso do comando CALL.

Exemplo :

```
CALL CALC_MEDIA(X,Y)
```

Seqüenciamento :

```
i : CALC_MEDIA(X,Y);      l = i+1;
```

b. functions :

As chamadas são feitas dentro de expressões.

Exemplo :

```
x = y + fx(t)
```

Seqüenciamento :

```
i     : var_fx = fx(t);  l = i+1;
i + 1 : x = y + var_fx;  l = i+2;
```

c. statement functions

São macros locais a um subprograma que são declaradas no início do corpo do subprograma. As macros são substituídas pelas suas referências nas expressões quando o programa é compilado; o seqüenciamento trata-as da mesma forma.

Exemplo :

```
FN(X,Y) = SIN(X)**2 - COS(Y)**2
...
W = TETA *  FN(X,Y)
```

Seqüenciamento :

```
i     : var_cos = cos(y)**2;        l=i+1;
i + 1 : var_sin = sin(x)**2;        l=i+2;
i + 2 : var_fn = var_sin-var_cos;   l=i+3;
i + 3 : w = teta*var_fn;            l=i+4;
```

Deve-se observar que para obter todos os nós do tipo **CALL**, durante o processo de seqüenciamento as expressões em que eles ocorrem devem ser particionadas e, se necessário, usar variáveis auxiliares para os valores de funções (como ocorre nos exemplos anteriores).

APENDICE C

Seqüenciamentos para as Estruturas de Controle da Linguagem C

Neste apêndice são relacionadas as principais estruturas de controle da linguagem de programação **C** e seus **seqüenciamentos**. Para obter a relação de estruturas de controle foi utilizado o livro "C - a Linguagem de Programação" [KERNIGHAN, 86] e as especificações do gerador de "parser" do **PCC** (Portable C Compiler [JOHNSON, 84-a]). Os **seqüenciamentos** foram estabelecidos de acordo com a Tabela 2.2. (**Tabela de Transformações**).

1. **Composição** : um conjunto de comandos com execução seqüencial, ou seja, um bloco de comandos, delimitado ou não por "{" e "}", é considerado uma composição.

Para a seguinte composição :

```
b = 1;
a+= 2;
```

Tem-se o seguinte seqüenciamento :

```
i   : B = 1;          1 = i + 1;
i+1 : A = A + 2;      1 = i + 2;
```

ou para :

```
{
     ++x;
     ++y;
}
```

tem-se :

```
i   : ++x;     1 = i+1
i+1 : ++y;     1 = i+2
```

2. **Seleção** : vários comandos em C representam seleções.

Na linguagem temos o comando *"IF"* com ou sem *"ELSE"*. Se o resultado do teste do predicado é verdadeiro a composição associada à parte "**then**" é executada, caso contrário é executada a composição correspondente à parte "**else**".

No exemplo a seguir, as composições são representadas como um comando seqüencial simples qualquer; denominado, para efeito de ilustração, por *<stmthen>* e *<stmelse>*.

Comando original :

```
if < condição >  <stmthen>;
else             <stmelse>;
...
```

Seqüenciamento :

```
i     : if <condição>    l = i + 1
        else             l = i + 3
i + 1 : <stmthen> ;      l = i + 2
i + 2 :                  l = i + 5
i + 3 : <stmelse> ;      l = i + 4
i + 4 :                  l = i + 5;
i + 5 : ...
```

O comando "**switch**" permite realizar uma seleção múltipla que é decomposta em uma cadeia de seleções simples, ou seja, forma uma composição de "**if's**" (ou nós do tipo OR). O comando pode utilizar qualquer composição de comandos em seus blocos, que são delimitados por "{" e "}", e/ou pelos seus rótulos *"case <valor-inteiro>:"* e *"default:"* [KERNIGHAN, 88].

No exemplo a seguir, é utilizada, para efeito de ilustração, a representação de um comando qualquer por *<stm_j_k>*, onde *j* representa o número do bloco de comando e *k* o número do comando no bloco.

Comando original :

```
...
switch (a) {
     case 1 :
                <stm_1_0>;
                <stm_1_1>;
     case 2:
                <stm_2>;
                break;
     case 3 :
                {
                <stm_3_0>;
                <stm_3_1>;
                }
     default:
                <stm_default>;
}
...
```

Seqüenciamento :

```
i - 1  : ...
i      :  if ( a == 1 )    l = i + 1;
          else             l = i + 3;
i + 1  :  <stm_1_0>;       l = i + 2;
i + 2  :  <stm_1_1>;       l = i + 3;
i + 3  :  if ( a == 2 )    l = i + 4;
          else             l = i + 6;
i + 4  :  <stm_2>;         l = i + 5;
i + 5  :                   l = i + 10;
i + 6  :  if ( a == 3 )    l = i + 7;
          else             l = i + 9;
i + 7  :  <stm_3_0>;       l = i + 8;
i + 8  :  <stm_3_1>;       l = i + 9;
i + 9  :  <stm_4>;         l = i + 10
i + 10 :  ...
```

Deve-se notar que a **seqüência** de execução nas seleções múltiplas do comando "**switch**" não é exclusiva; se uma condição é verdadeira seu bloco será executado, no entanto, executar um bloco não implica a não execução do teste do próximo rótulo. O próximo teste não é realizado se e somente se ocorrer um desvio para fora do bloco correspondente ao **teste** anterior através do uso do comando "**break**".

Uma seleção pode estar presente em uma expressão; este tipo de expressão pode ser genericamente especificada por :

<e1> ? <e2> : <e3>

Onde <e1>, <e2> e <e3> são quaisquer expressões válidas para a linguagem **C**. Se o resultado da expressão <e1> for um valor diferente de zero, o resultado de toda a expressão será o resultado da expressão <e2>, caso contrário o resultado será o da expressão <e3>.

Uma expressão desse tipo é seqüenciada como uma seleção simples.

Comando original :

x = y + z ? a : d;

Seqüenciamento :

```
i     : if ( z )          l = i + 1;
        else              l = i + 2;
i + 1 : quest_1 = a;      l = i + 3;
i + 2 : quest_1 = d;      l = i + 3;
i + 3 : x = y + quest_1;  l = i + 4;
```

O tipo de **seqüenciamento** acima cria uma variável auxiliar *"quest_k"*; onde ***k*** é um inteiro cuja função é simplesmente diferenciar a variável se houver outras expressões condicionais no mesmo escopo do programa. A variável auxiliar *"quest_k"* guardará o valor da expressão final, isto é, da expressão que for executada dependendo do resultado do teste do predicativo.

3. **Iterações** : as iterações, ou laços de execução [KERNIGHAN, 88], são representadas pelas estruturas de controle constantes na **Tabela de Transformações**, que por sua vez são transformadas em "**do-while**".

a. "**while**"

No exemplo seguinte, *<e_cond>* corresponde ao predicativo do "**while**" e cada *<stm_j_k>* a um comando qualquer do laço.

Comando original :

```
while ( <e_cond> ) {
        <stm_1_0>;
        <stm_1_1>;
}
...
```

Seqüenciamento :

```
i     : if ( <e_cond> )          l = i + 1;
        else                     l = i + 4;
i + 1 : <stm_1_0>;               l = i + 2;
i + 2 : <stm_1_1>;               l = i + 3;
i + 3 : if ( <e_cond> )          l = i + 1;
        else                     l = i + 4;
i + 4 : ...
```

b. "**do-while-do**"

Neste caso, existe um bloco de comandos que deve ser executado antes do teste do valor do predicado para cada iteração. Esta estrutura de controle para a linguagem C pode ser escrita como no exemplo abaixo.

Comando original :

```
while ( a = a + 1,  j < 10 ) {
        vetor[a] = vetor[j] * vetor[a];
        j++;
}
...
```

O seqüenciamento mantém a equivalência de execução acima :

```
i     : a = a + 1;               l = i + 1;
i + 1 : if ( j < 10 )            l = i + 2;
        else                     l = i + 6;
i + 2 : vetor[a] = vetor[j] * vetor[a];
                                 l = i + 3;
i + 3 : j++;                     l = i + 4;
i + 4 : a = a + 1;               l = i + 5;
i + 5 : if ( j < 10 )            l = i + 2;
        else                     l = i + 6;
i + 6 : ...
```

c. "for"

Comando original :

```
for ( k = LIMINF ; k <= LIMITE ; k++ ) {
     vetor[k] *= vetor[k];
}
...
```

Seqüenciamento :

```
i     : k = LIMINF;              l = i + 1;
i + 1 : if ( k <= LIMITE )       l = i + 2;
        else                     l = i + 4;
i + 2 : vetor[k] *= vetor[k];    l = i + 3;
i + 3 : if ( k <= LIMITE )       l = i + 2;
        else                     l = i + 4;
i + 4 : ...
```

d. "do-while"

Comando original :

```
<stm_0>;
do {
     <stm_1_0>;
     <stm_1_1>;
} while ( <e_cond> );
...
```

Seqüenciamento :

```
i     : <stm_0>;           l = i + 1;
i + 1 : <stm_1_0>;         l = i + 2;
i + 2 : <stm_1_1>;         l = i + 3;
i + 3 : if ( <e_cond> )    l = i + 1;
        else               l = i + 4;
i + 4 : ...
```

4. Desvios : Os desvios, na linguagem C, são representados pelos comandos "goto", "break", "continue" e "return".

a. "goto"

Bloco de comandos original :

```
rotulo_1 :    <stm_1_0>;
              <stm_1_1>;
              <stm_1_2>;
              <stm_1_3>;
              if ( <e_cond> )
                        goto rotulo_1;
```

Seqüenciamento :

```
i     : <stm_1_0>;          l = i + 1;
i + 1 : <stm_1_1>;          l = i + 2;
i + 2 : <stm_1_2>;          l = i + 3;
i + 3 : <stm_1_3>;          l = i + 4;
i + 4 : if ( <e_cond> )     l = i + 5
          else              l = i + 6;
i + 5 :                     l = i;
i + 6 : ...
```

b. "**break**"

E usado dentro de um laço de execução, bloco de comandos em uma iteração. Ou, também é usado em um bloco de comandos de um rótulo do comando "**switch**". Este desvio leva o fluxo de execução para o primeiro comando após o fim do bloco, como no exemplo a seguir.

Bloco original :

```
do {
     <stm_1_0>;
     <stm_1_1>;
     if ( <e_cond_1> )
               break;
} while ( <e_cond_2> );
<stm_2_0>;
```

Seqüenciamento :

```
i     : <stm_1_0>;          l = i + 1;
i + 1 : <stm_1_1>;          l = i + 2;
i + 2 : if ( <e_cond_1> )   l = i + 3;
          else              l = i + 4;
i + 3 :                     l = i + 7;
i + 4 :                     l = i + 5;
i + 5 : if ( <e_cond_2> )   l = i;
          else              l = i + 6;
i + 6 : <stm_2_0>;          l = i + 7;
```

c. "continue"

E usado dentro de um laço de execução, porém ele desvia o fluxo de execução para a próxima iteração; o "**continue**" quebra a execução de uma determinada iteração. Se, no exemplo anterior, fosse usado um "**continue**" em lugar do "**break**", o **seqüenciamento** mostraria outro fluxo de execução, como no exemplo a seguir.

Bloco original :

```
do {
      <stm_1_0>;
      <stm_1_1>;
      if ( <e_cond_1> )
                  continue;
} while ( <e_cond_2> );
<stm_2_0>;
```

Seqüenciamento :

```
i     : <stm_1_0>;          l = i + 1;
i + 1 : <stm_1_1>;          l = i + 2;
i + 2 : if ( <e_cond_1> )   l = i + 3;
        else                l = i + 4;
i + 3 :                     l = i + 5;
i + 4 :                     l = i + 5;
i + 5 : if ( <e_cond_2> )   l = i;
        else                l = i + 6;
i + 6 : <stm_2_0>;          l = i + 7;
```

E, se em lugar do "**do-while**" tivesse sido usado um "**for**", como no bloco abaixo, o seqüenciamento seria :

Bloco original :

```
for ( <e_0> ; <e_cond_2> ; <e_n> ) {
      <stm_1_0>;
      <stm_1_1>;
      if ( <e_cond_1> )
            continue;
}
<stm_2_0>;
```

Seqüenciamento :

```
i     : <e_0>; l = i + 1;
i + 1 : if ( <e_cond_2> )    l = i + 2;
        else                 l = i + 8;
i + 2 : <stm_1_0>;           l = i + 3;
i + 3 : <stm_1_1>;           l = i + 4;
i + 4 : if ( <e_cond_1> )    l = i + 5;
        else                 l = i + 6;
i + 5 :                      l = i + 7;
i + 6 : <e_n>;               l = i + 7;
i + 7 : if ( <e_cond_2> )    l = i + 2;
        else                 l = i + 8;
i + 8 : <stm_2_0>;           l = i + 9;
```

d. "return"

Este comando também pode ser considerado um desvio, uma vez que ele é usado para encerrar a execução de uma função. Em alguns casos, o uso do "**return**" acarreta a inserção de mais de um ponto de saída no escopo de uma mesma função, como no exemplo a seguir.

Exemplo :

```
<nome_função>  (<lista_parâmetros>)
<declarações_parâmetros>;
{
     <declarações_locais>;
          <stm_1_0>;
          if ( <e_cond_1> ) {
               <stm_2_0>;
               return(<expressão_1>);
          }
          <stm_1_1>;
          return(<expressão_2>);
}
```

A função acima tem dois pontos de saída; para obter a sua **Primeira Forma Normal** é necessário garantir que haja apenas um ponto de saída para cada função.

Uma alternativa é criar uma variável auxiliar para o comando "**return**" de uma função, que guarde o valor da expressão a ser retornada e para cada "**return**" a próxima **seqüência** ser a de número 0; como no **seqüenciamento** a seguir.

```
i     : <stm_1_0>;                    l = i + 1;
i + 1 : if ( <e_cond_1> )             l = i + 2;
        else                          l = i + 5;
i + 2 : <stm_2_0>;                    l = i + 3;
i + 3 : FNC_RETURN = <expressão_1>;   l = 0;
i + 4 :                               l = i + 5;
i + 5 : <stm_1_1>;                    l = i + 6;
i + 6 : FNC_RETURN = <expressão_2>;   l = 0;
```

O comando "**return**" será colocado após o nó do tipo **LOOP** da **normalização**, isto é, será o comando sucessor ao "**do-while (l>0)**" (no início do Capítulo 3 é dado um exemplo deste caso).

5. Subprogramas :

Os subprogramas na linguagem **C** correspondem a chamadas às funções do programa e/ou bibliotecas de funções.

As funções sempre estão contidas em expressões quaisquer da linguagem [JOHNSON,84] e seus resultados são utilizados na expressão. Como para o processo de decomposição é necessário separar chamadas de funções em nós do tipo **CALL**, as expressões serão quebradas para

cada chamada de função. Para realizar os **seqüenciamentos** serão criadas novas variáveis associadas aos nomes e ocorrências das chamadas de funções.

Exemplo :

```
x = fx(a+b) - fy(o) + 3 + fw(j,k) * y;
```

Na expressão acima, temos três referências a subprogramas. Para separá-las em **seqüências** que contenham nós do tipo **CALL** assume-se que as chamadas de funções têm precedência sobre o restante da expressão (como no compilador **C** portável da Bell Laboratories [JOHSON, 84-a]); o **seqüenciamento** é o seguinte :

```
i     : fx_1 = fx(a+b);                    l = i + 1;
i + 1 : fy_2 = fy(o);                      l = i + 2;
i + 2 : fw_3 = fw(j,k);                    l = i + 3;
i + 3 : x = fx_1 - fy_2 + 3 + fw_3 * y;
                                           l = i + 4;
```

Não é necessário criar uma variável auxiliar para o **seqüenciamento** se e somente se a função para a qual existe uma chamada não retorna valor, isto é, ela é do tipo "**void**" [KERNIGHAN, 88].

Por exemplo :

```
printf("EXEMPLO\n");
```

A chamada para a função "**printf**" não fornece um valor de retorno, logo seu **seqüenciamento** é :

```
i : printf("EXEMPLO\n");    l = i + 1;
```

Observação :

Se uma chamada de função ocorre em uma expressão de um laço, a chamada à função deve ser repetida na última **seqüência** do laço; por exemplo :

```
while ( f1(x) < b ) { x = (a++) * b; }
```

Seqüenciamento :

```
1 : var_f1 = f1(x);          l = 2;
2 : if ( var_f1(x)  < b )    l = 3
    else                     l = 6;
3 : x = (a++) * b;           l = 4;
4 : var_f1 = f1(x);          l = 5;  /* última seqüência do laço */
5 : if ( var_f1 < b )        l = 3
    else                     l = 6;
6 : ...     /* fim do laço */
```

APENDICE D

Referências ao Módulo SPLINE/NTIA — Original e Estruturado

1. Resultados Completos da Execução de SPLINE e SPLINES :

1.a. Arquivo com dados originais, nome "serie.dad" :

```
 1;150.57
 2;160.32
 3;169.0
 4;179.78
 5;123.78
 6;130.22
 7;138.18
 8;149.78
 9;167.85
10;179.83
11;143.13
12;144.43
13;152.82
14;153.07
15;155.0
16;177.9
17;129.65
18;131.92
19;148.3
20;157.72
21;173.78
22;198.88
23;155.13
24;156.17
25;159.97
26;163.83
27;180.78
28;193.6
29;127.38
30;125.45
31;125.15
32;143.55
33;163.73
34;172.47
35;119.68
36;120.48
```

1.b. Resultado da execução do Módulo SPLINE, Software NTIA :

SPLINE - ANTES

obs	*mes*	*tempo*
1	*36.0000*	*120.4800*
2	*35.5000*	*113.1120*
3	*35.0000*	*119.6800*
4	*34.5000*	*146.8829*
5	*34.0000*	*172.4700*
6	*33.5000*	*174.8140*
7	*33.0000*	*163.7300*
8	*32.5000*	*153.3397*
9	*32.0000*	*143.5500*
10	*31.5000*	*132.4596*
11	*31.0000*	*125.1500*
12	*30.5000*	*125.4818*
13	*30.0000*	*125.4500*
14	*29.5000*	*119.9545*
15	*29.0000*	*127.3800*
16	*28.5000*	*161.4301*
17	*28.0000*	*193.6000*
18	*27.5000*	*195.4213*
19	*27.0000*	*180.7800*
20	*26.5000*	*169.6283*
21	*26.0000*	*163.8300*
22	*25.5000*	*161.0153*
23	*25.0000*	*159.9700*
24	*24.5000*	*159.3541*
25	*24.0000*	*156.1700*
26	*23.5000*	*150.3408*
27	*23.0000*	*155.1300*
28	*22.5000*	*179.1266*
29	*22.0000*	*198.8800*
30	*21.5000*	*192.1753*
31	*21.0000*	*173.7800*
32	*20.5000*	*162.6758*
33	*20.0000*	*157.7200*
34	*19.5000*	*153.5815*
35	*19.0000*	*148.3000*
36	*18.5000*	*141.0182*
37	*18.0000*	*131.9200*
38	*17.5000*	*123.8995*
39	*17.0000*	*129.6500*
40	*16.5000*	*156.1726*
41	*16.0000*	*177.9000*
42	*15.5000*	*171.4813*
43	*15.0000*	*155.0000*
44	*14.5000*	*150.3298*
45	*14.0000*	*153.0700*

SPLINE - ANTES

obs	mes	tempo
46	13.5000	154.2406
47	13.0000	152.8200
48	12.5000	149.5701
49	12.0000	144.4300
50	11.5000	139.0976
51	11.0000	143.1300
52	10.5000	162.3557
53	10.0000	179.8300
54	9.5000	179.0247
55	9.0000	167.8500
56	8.5000	157.7394
57	8.0000	149.7800
58	7.5000	142.9552
59	7.0000	138.1800
60	6.5000	135.5835
61	6.0000	130.2200
62	5.5000	120.5556
63	5.0000	123.7800
64	4.5000	152.1890
65	4.0000	179.7800
66	3.5000	180.8257
67	3.0000	169.0000
68	2.5000	162.7630
69	2.0000	160.3200
70	1.5000	156.2111
71	1.0000	150.5700

1.c.Resultado da execução do Módulo SPLINES, SPLINE estruturado :

SPLINE - DEPOIS

obs	*mes*	*tempo*
1	*36.0000*	*120.4800*
2	*35.5000*	*113.1120*
3	*35.0000*	*119.6800*
4	*34.5000*	*146.8829*
5	*34.0000*	*172.4700*
6	*33.5000*	*174.8140*
7	*33.0000*	*163.7300*
8	*32.5000*	*153.3397*
9	*32.0000*	*143.5500*
10	*31.5000*	*132.4596*
11	*31.0000*	*125.1500*
12	*30.5000*	*125.4818*
13	*30.0000*	*125.4500*
14	*29.5000*	*119.9545*
15	*29.0000*	*127.3800*
16	*28.5000*	*161.4301*
17	*28.0000*	*193.6000*
18	*27.5000*	*195.4213*
19	*27.0000*	*180.7800*
20	*26.5000*	*169.6283*
21	*26.0000*	*163.8300*
22	*25.5000*	*161.0153*
23	*25.0000*	*159.9700*
24	*24.5000*	*159.3541*
25	*24.0000*	*156.1700*
26	*23.5000*	*150.3408*
27	*23.0000*	*155.1300*
28	*22.5000*	*179.1266*
29	*22.0000*	*198.8800*
30	*21.5000*	*192.1753*
31	*21.0000*	*173.7800*
32	*20.5000*	*162.6758*
33	*20.0000*	*157.7200*
34	*19.5000*	*153.5815*
35	*19.0000*	*148.3000*
36	*18.5000*	*141.0182*
37	*18.0000*	*131.9200*
38	*17.5000*	*123.8995*
39	*17.0000*	*129.6500*
40	*16.5000*	*156.1726*
41	*16.0000*	*177.9000*
42	*15.5000*	*171.4813*
43	*15.0000*	*155.0000*
44	*14.5000*	*150.3298*
45	*14.0000*	*153.0700*

SPLINE - DEPOIS

obs	mes	tempo
46	13.5000	154.2406
47	13.0000	152.8200
48	12.5000	149.5701
49	12.0000	144.4300
50	11.5000	139.0976
51	11.0000	143.1300
52	10.5000	162.3557
53	10.0000	179.8300
54	9.5000	179.0247
55	9.0000	167.8500
56	8.5000	157.7394
57	8.0000	149.7800
58	7.5000	142.9552
59	7.0000	138.1800
60	6.5000	135.5835
61	6.0000	130.2200
62	5.5000	120.5556
63	5.0000	123.7800
64	4.5000	152.1890
65	4.0000	179.7800
66	3.5000	180.8257
67	3.0000	169.0000
68	2.5000	162.7630
69	2.0000	160.3200
70	1.5000	156.2111
71	1.0000	150.5700

2. Código Fonte de "token_var" - Antes e Após Estruturação :

2.a. Código fonte da função "token_var" antes da estruturação :

```
void token_var()
{
     int i;
     char nome[10];
     double min;
     double max;
     char   tipo;

     if ((c=getoken(token)) != 256 )
          erroc(1);
     ++nv;
     if (nv > 1)
          erroc(2);
     while(c != 258 ) {
          if (c != 256    && c != 258    )
               erroc(1);

          for(i=0; i<npar; ++i)
               if(strcmpi(obs1[i],token) == 0)
                    erroc(4);
          if ((i = veri_var(fd,token)) < 0)
               erroc(6);
          pega_var(fd,i,nome,&tipo,(union MINS *) &min,
                    (union MAXS *) &max) ;
          if (tipo == 'c')
               erroc(7);
          nomevar[nvar] = npar;
          strcpy(obs1[npar++],token);
          posvar[nvar++] = i;
          c = getoken(token);
     }
}
....
```

2.b. Código fonte da função "token_var" após a estruturação :

A função "token_var" tem seu código fonte estruturado no arquivo "token_va.FNR", que é o seguinte :

```
/* Process : Reduction
   Code Unit : token_va.FNR
   Process Date  : Wed Sep 16 09:42:45 1992  */

void token_var ( ){
int i;
char nome [ 10 ];
double min;
double max;
char tipo;
int  FNC_getoken_0;
int  FNC_strcmpi_4;
int  FNC_veri_var_6;
int  FNC_pega_var_8;
char  * FNC_strcpy_10;
int  FNC_getoken_11;

      FNC_getoken_0 = getoken(token);
      if ((c = FNC_getoken_0)!=256) {
           erroc(1);
      }
      ++nv;
      if (nv>1) {
           erroc(2);
      }
      if (c!=258) {
           do {
                if (c!=256&&c!=258) {
                     erroc(1);
                }
                i = 0;
                if (i<npar) {
                     do {
                          FNC_strcmpi_4 = strcmpi(obs1[i],token);
                          if (FNC_strcmpi_4==0) {
                               erroc(4);
                          }
                          ++i;
                      } while (i<npar) ;
                }
                FNC_veri_var_6 = veri_var(fd,token);
                if ((i = FNC_veri_var_6)<0) {
                     erroc(6);
                }
                FNC_pega_var_8 = pega_var(fd,i,nome,&tipo,(union
                          MINS  *)&min,(union MAXS  *)&max);
                if (tipo=='c') {
                     erroc(7);
                }
                nomevar[nvar] = npar;
                FNC_strcpy_10 = strcpy(obs1[npar++],token);
                posvar[nvar++] = i;
                FNC_getoken_11 = getoken(token);
                c = FNC_getoken_11;
           } while (c!=258) ;
      }
}
```

3. Código Fonte de "rhs" - Antes e Após Estruturação :

3.a. Código fonte de "rhs" original :

```
float rhs(i)
{
     int i_;
     double zz;
     i_ = i==n-1?0:i;
     zz = (y.val[i]-y.val[i-1])/(x.val[i]-x.val[i-1]);
     return(6*((y.val[i_+1]-y.val[i_])/(x.val[i+1]-x.val[i])
                    - zz));
}
```

3.b. Código fonte de "rhs" após estruturação :

```
/*
Process : Reduction
Code Unit : rhs.FNR
Process Date  : Wed Sep 16 09:44:08 1992
*/

float rhs ( i ){

int i_;

double zz;
float  FNC_RETURN;
int  FNC_QUEST_0;

     if (i==n-1) {
          FNC_QUEST_0 = 0;
     }
     else {
          FNC_QUEST_0 = i;
     }
     i_ = FNC_QUEST_0;
     zz = (y.val[i]-y.val[i-1])/(x.val[i]-x.val[i-1]);
     FNC_RETURN = (6*((y.val[i_+1]-y.val[i_])/(x.val[i+1]-
                         x.val[i])-zz));
     return(FNC_RETURN);
}
```

4. Código Fonte de "semantica" - Antes e Após Estruturação :

4.a. Código fonte original da função "semantica" :

```
void semantica()
{
      extern void exit();
      int teste;
      setjmp(go);
      inicializa1();
      teste=0;

      for(;;) {
            c = getoken(token);

            if ((c==(-1)) || (c==266 ))
                  executa = 1;

            if (executa == 1)
                  break;

            if (strcmpi(token,"cancela") == 0)
                  exit(1);

            teste=descobre();

            if (teste == -1)
                  erroc(4);

            switch(temp) {

            case 'v':
                  token_var();
                  break;
            default:
                  erroc(3);
                  break;
            }
      }
      if (nvar < 1 ) {
            erroc(5);
      }
}
```

4.b. Código fonte da função "semantica" após estruturação :

```
/* Process : Reduction
   Code Unit : semantic.FNR
   Process Date  : Wed Sep 16 09:41:47 1992   */

void semantica ( ){

extern void exit ( );

int teste;
int  FNC_setjmp_0;
int  FNC_getoken_2;
int  FNC_strcmpi_3;
int  FNC_descobre_5;
int  FNC_L;

      FNC_setjmp_0 = setjmp(go);
      inicializa1();
      teste = 0;
      if (1) {
           FNC_L = 5;
      }
      else {
           FNC_L = 28;
      }
      do {
           if (FNC_L==5) {
                FNC_getoken_2 = getoken(token);
                c = FNC_getoken_2;
                if ((c==(-1))||(c==266)) {
                     executa = 1;
                }
                if (executa==1) {
                     FNC_L = 28;
                }
                else {
                     FNC_strcmpi_3 = strcmpi(token,"cancela");
                     if (FNC_strcmpi_3==0) {
                          exit(1);
                     }
                     FNC_descobre_5 = descobre();
                     teste = FNC_descobre_5;
                     if (teste==-1) {
                          erroc(4);
                     }
                     if ('v' == temp) {
                          token_var();
                     }
                     else {
                          erroc(3);
                     }
                     if (1) {
                          FNC_L = 5;
                     }
                     else {
                          FNC_L = 28;
                     }
```

```
                }
            }
            if (FNC_L==28) {
                if (nvar<1) {
                    erroc(5);
                }
                FNC_L = 0;
            }
        } while (FNC_L > 0) ;
}
```